
DISEÑO E IMPLEMENTACIÓN DE UNA CAPA DE SOFTWARE PARA EL CONTROL DE ROBOTS MEDIANTE PLAYER/STAGE

PROYECTO FIN DE CARRERA

Preparado por: **Tomás Rebollo Balaguer**

Fecha: **Enero de 2010**

Directores del proyecto: **Jesús García Herrero**
 José Luís Guerrero Madrid



ÍNDICE DE CONTENIDOS

1 INTRODUCCIÓN	6
1.1 OBJETIVOS Y MOTIVACIONES.....	8
2.1 PLANTEAMIENTO DEL PROYECTO.....	9
2 ESTADO DEL ARTE.....	11
2.1 VEHÍCULOS NO TRIPULADO (UAV Y UGV).....	12
2.2 SENSORES INERCIALES Y DE POSICIONAMIENTO	14
2.2.1 <i>Sensores Inerciales</i>	14
2.2.2 <i>Sensores de Posicionamiento</i>	16
2.3 SISTEMAS DE FUSION DE DATOS DE SENSORES.....	18
2.3.1 <i>Niveles de fusión</i>	19
2.4 SISTEMAS DE CONTROL DE ROBOTS	22
2.4.1 <i>Sistemas de lazo abierto</i>	22
2.4.2 <i>Sistemas de lazo cerrado</i>	23
3 EL ROBOT GUARDIÁN	25
3.1 CARACTERÍSTICAS DEL ROBOT.....	26
3.2 FUNCIONAMIENTO Y CONTROL	28
4 PLAYER/STAGE.....	30
4.1 NECESIDAD DE UTILIZAR HERRAMIENTAS SOFTWARE	31
4.2 PLAYER.....	32
4.2.1 <i>¿Qué es Player?</i>	32
4.2.2 <i>Player y la arquitectura cliente-servidor</i>	32
4.2.3 <i>Interfaces, drivers y dispositivos en Player</i>	33
4.2.4 <i>Ficheros de configuración del servidor Player</i>	36
4.2.5 <i>Programas cliente en Player</i>	39
4.2.6 <i>Librerías de Player</i>	42
4.3 STAGE.....	43

4.3.1 ¿Qué es Stage?.....	43
4.3.2 Proceso de simulación de Stage.....	44
4.3.3 Configuración de Stage.....	45
5 DESARROLLO.....	49
5.1 PUNTO DE PARTIDA	49
5.1.2 Necesidades y objetivos	50
5.1.1 El problema de la asincronía	50
5.2 DISEÑO DE LA CAPA DE SOFTWARE	52
5.2.1 Estructura de Datos Dinámica.....	53
5.2.2 Métodos de acceso a los datos.....	54
5.3 IMPLEMENTACIÓN DE LA CAPA DE SOFTWARE.....	55
5.3.1 Diagrama de clases.....	56
5.3.2 Detalles de la implementación.....	57
6 EXPERIMENTACIÓN.....	67
6.1 INTRODUCCIÓN Y PREPARACIÓN DE LAS PRUEBAS	68
6.2 PRUEBAS CON TRAYECTORIAS RECTILÍNEAS	69
Prueba 1.....	69
Prueba 2.....	73
6.3 PRUEBAS CON TRAYECTORIA CURVAS.....	77
Prueba 1.....	77
Prueba 2.....	79
Prueba 3.....	82
Prueba 4.....	84
7 CONCLUSIONES	90
8 BIBLIOGRAFÍA.....	94
9 ANEXOS.....	97
ANEXO A: GESTIÓN DEL PROYECTO	98
Planificación de las tareas.....	98
Gestión económica del proyecto.....	102
ANEXO B: GUÍA DE INSTALACIÓN DE PLAYER/STAGE	104

<i>Paso 1</i>	104
<i>Paso 2</i>	104
<i>Paso 3</i>	106
ANEXO C: MANUAL DE USUARIO DEL MIDDLEWARE	111
<i>Programa de control del robot</i>	112
<i>Lista de datos de sensores</i>	117
<i>Otros aspectos del control del robot</i>	119

INTRODUCCIÓN

CAPITULO

En la actualidad, el uso de los robots teledirigidos y autónomos está experimentando un crecimiento sin igual, sobre todo en el ámbito militar, debido a sus múltiples posibilidades de uso y al abaratamiento de la tecnología necesaria para su construcción.

Gracias al avance tecnológico de los últimos años, hoy en día es posible construir máquinas con la suficiente potencia de procesamiento, dotadas de dispositivos sensoriales, con facilidades de comunicación a larga distancia y con una precisión en la realización de diferentes acciones bastante elevada, que pueden llegar a realizar tareas sin la presencia de una persona y con un cierto grado de autonomía y suficiencia, que en algunos casos es total. Existen varios tipos de robots que se diferencian en el grado de autonomía que poseen y en el tipo de entorno en el que son utilizados. En este proyecto se manejan en concreto conceptos relacionados con los vehículos no tripulados [1] en el ámbito aéreo y terrestre, cuyas siglas en inglés corresponden a UAV (Unmanned Aerial Vehicle) [2] y UGV (Unmanned Ground Vehicle) [2] respectivamente.

Toda máquina robótica, sobre todo los UAV y los UGV, necesita estar bien programada para que pueda realizar las tareas deseadas de la forma correcta. Cuanto más avanzado es el robot, más difícil es realizar los programas de control y si se trata de robots autónomos, dicha programación es el punto más crítico, ya que es tarea del desarrollador que los distintos sensores y actuadores se comuniquen correctamente para que el robot sea capaz de tomar las decisiones correctas.

Cuando se trata de robots autónomos, las acciones que estos llevan a cabo vienen precedidas de la toma de decisiones. Las decisiones que toman éste tipo de robots,

dependen principalmente de los datos obtenidos a partir de los distintos tipos de sensores, como GPS, IMU, etc. Analizando los datos capturados por los sensores, el robot decide qué acción debe de realizar en un momento dado. Dentro de las múltiples acciones y tareas que puede llevar a cabo un robot autónomo, la navegación es un punto fundamental, ya que establece cómo y de qué manera se mueve el robot [3]. Es tal la importancia de la navegación, que se han desarrollado métodos para aumentar la precisión de los datos obtenidos por los sensores, de forma que el robot pueda conocer su posición actual con una precisión bastante elevada. Estos métodos se conocen como sistemas de fusión de datos [4], los cuales se definen a grandes rasgos por utilizar la información de varios sensores, generalmente de tipo GPS o INS, para obtener la posición con mayor precisión, realizando una integración matemática, o para aumentar la fiabilidad mediante el uso de sensores redundantes.

Los sistemas de fusión de datos de sensores, se utilizan para procesar varios datos de diferentes sensores (o de sensores redundantes) al mismo tiempo, consiguiendo disminuir el error que poseen los datos (todos los sensores tienen un margen de error en las mediciones) y por consiguiente, aumentando la precisión de los mismos [4]. Generalmente, estos sistemas de fusión de datos se utilizan en la navegación, ya que saber la posición con la mayor precisión posible es crucial para la estabilidad y el éxito del robot. Por tanto, suelen utilizarse en conjunto con sensores GPS y sensores inerciales (IMU, INS, etc.). Dada la cantidad de datos que estos sistemas han de manejar, generalmente en tiempo real, las necesidades de procesamiento son muy elevadas, por lo que la programación de estos sistemas ha de ser lo más sencilla y eficiente posible. Cabe destacar que estos sistemas se suelen utilizar en entornos de tiempo real, por lo que la importancia de disponer de los datos de los sensores de forma rápida y fiable es crucial.

Desarrollar un robot autónomo es altamente complicado. Afortunadamente, en la actualidad es posible encontrar diversas soluciones comerciales o proyectos de código abierto, centrados en la programación de robots, los cuales ofrecen facilidades en esta complicada tarea. Gracias a los módulos y a las librerías proporcionadas por alguna de estas soluciones, el desarrollador evita la programación de aspectos de bajo nivel, como la programación de los controladores que comuniquen los distintos dispositivos con el sistema del robot.

La programación de robots se refiere a la implementación en un lenguaje de programación del comportamiento que ha de tener el robot, de las acciones que va a llevar a cabo, de cómo va a realizarlas, etc. En definitiva, se trata de una pieza de software (a veces incluso de hardware) en la que se define el control del robot, así como la lógica del mismo, haciendo uso de todos los medios disponibles, como los sensores o los actuadores. Generalmente, los programas de control constan de un bucle principal en el que se procesan los datos de los sensores, se aplica un razonamiento y se ejecutan las acciones, y así sucesivamente.

La proliferación de las herramientas centradas en la programación de robots, ha hecho que muchos usuarios particulares y empresas se hayan decantado por incluirlas a la hora de realizar sus proyectos con máquinas reales. Este es el caso de la empresa Robotnik Automation S.L., cuyo robot Guardián se basa en el uso de la plataforma libre

Player/Stage [5] [6], plataforma en auge que ofrece un completo conjunto de librerías y facilidades para la programación de robots autónomos.

La herramienta Player/Stage proporciona los medios necesarios para comunicarse con multitud de dispositivos sensores de diferentes fabricantes, por lo que resulta una herramienta muy versátil, ya que permite instalar nuevos sensores sin tener que realizar grandes modificaciones en el proyecto.

A raíz de las numerosas posibilidades y facilidades que ofrecen herramientas como Player/Stage y de las necesidades actuales de los sistemas de fusión de datos, sobre todo en entornos de tiempo real en los que los datos de los sensores han de ser accesibles con rapidez y de forma fiable, surge la motivación de realizar un proyecto como este, en el cual se pretende llevar a cabo una primera toma de contacto con un robot real con múltiples dispositivos sensores, como es el robot Guardián, al mismo tiempo que se intenta implementar una pieza de software genérica que comunique los sistemas de fusión de datos, con los sensores del robot de una forma sencilla y transparente, con el fin de adaptarse a los sistemas de fusión de datos ya existentes y aislarlos de las tareas de más bajo nivel, como lo son la comunicación con el hardware del robot, proporcionándoles al mismo tiempo diferentes métodos de acceso a los datos de los sensores, de forma rápida y fiable, gracias a la inclusión de una lista genérica en la que se almacenan todos los datos.

1.1 OBJETIVOS Y MOTIVACIONES

El inicio de un proyecto como este, motivado por las necesidades de los algoritmos de fusión de datos de sensores y por realizar la primera toma de contacto con una máquina real, requiere un esfuerzo previo de estudio y comprensión de múltiples aspectos, debido a la complejidad que supone iniciar el desarrollo de un sistema de control basado en la navegación sobre un robot real y apoyado en una plataforma concebida como un pequeño sistema operativo para robots, como la reciente Player/Stage.

Los objetivos que se han establecido al inicio de este proyecto pasan por el estudio de la plataforma Player/Stage para el control de robots y sus dispositivos, así como para realizar simulaciones en entornos virtuales; y por la comprensión del estado actual de los sistemas autónomos y los sistemas de fusión de datos de sensores como el GPS o los sensores inerciales. Finalmente, a partir de los conocimientos adquiridos sobre estos conceptos, se pretende construir una capa de software básica, apoyada en la citada plataforma Player/Stage, que permita interactuar con el robot Guardián de una manera sencilla, gracias a la abstracción de los mecanismos de comunicación entre el robot y sus dispositivos, propiciando de este modo una fácil implantación de sistemas de más alto nivel, como los citados sistemas de fusión de datos de sensores y posteriormente algoritmos de inteligencia artificial que permitan al robot Guardián ser un robot totalmente autónomo.

2.1 PLANTEAMIENTO DEL PROYECTO

Los objetivos de este proyecto son bastante claros y específicos, por lo que una vez se han definido dichos objetivos, el siguiente paso consiste en identificar las tareas o fases que han de llevarse a cabo para lograr alcanzar los objetivos marcados. Este documento recoge todas y cada una de las distintas fases por las que ha pasado el proyecto, desde su concepción, pasando por el estudio previo de las tecnologías existentes, hasta llegar a la implementación de un componente software que cubra las necesidades.

Las diferentes fases o etapas por las que ha ido pasando el proyecto hasta cumplir con los objetivos marcados, corresponden a las siguientes:

- ⌘ **Estado del arte:** Estudio del entorno tecnológico y de las bases teóricas sobre las que se apoya el proyecto.
- ⌘ **El robot Guardián:** Descripción del robot Guardián, especificando todas sus características y sus posibilidades, así como sus defectos e inconvenientes.
- ⌘ **Player/Stage:** Estudio y análisis de la plataforma Player/Stage para el control de robots.
- ⌘ **Desarrollo:** Desarrollo de la capa de software intermedia que cubra las necesidades del proyecto, las cuales abarcan el acceso en tiempo real a los datos recogidos por los distintos tipos de sensores que incorpora el robot Guardián, la creación de una estructura de tipo lista que permita almacenar los datos de los sensores de forma genérica y la implementación de diversos métodos de acceso a dicha lista para un uso más flexible.
- ⌘ **Experimentación:** Pruebas realizadas con el simulador Stage y en condiciones reales con el robot Guardián. Se analizan y comentan los resultados de cada prueba.
- ⌘ **Conclusiones:** Finalmente se estudia y analiza todos los aspectos del proyecto y se comentan las conclusiones y puntos de vista sobre el proyecto, así como se realiza una breve descripción de los trabajos a realizar en el futuro.

Este documento recoge todas y cada una de las fases por las que ha pasado el proyecto, a las que se les dedica un apartado específico y que demuestran la evolución del proyecto hasta su terminación, explicando las conclusiones obtenidas en cada una de ellas.

ESTADO DEL ARTE

CAPITULO

Este proyecto, como muchos otros, no pretende establecer una nueva base teórica, sino que se apoya en una serie de conceptos abiertamente aceptados, a partir de los cuales se intenta avanzar hacia los objetivos propuestos. Es necesario hacer un repaso del estado actual de las tecnologías de las que se hace uso, así como de los conceptos sobre los que se apoya este proyecto.

Principalmente, este proyecto parte de la base de los robots autónomos y de la importancia de la fiabilidad de los datos recogidos por los diferentes sensores. Los robots autónomos tienen la característica de que no son tripulados por ninguna persona. Pueden ser controlados remotamente por una persona (aparatos de radio control, etc.) o por el contrario tienen la capacidad de actuar por sí solos, tomando las decisiones convenientes en cada situación, sin la ayuda de ninguna persona. La toma de decisiones está condicionada por los datos recogidos de los diferentes sensores que pueda tener un robot autónomo. Es por este motivo por el que la fiabilidad de los datos de los sensores y el conocimiento detallado de los propios dispositivos es un punto tan importante, ya que es muy posible que los dispositivos sensores necesiten correcciones debido a los pequeños errores o desviaciones en las mediciones, con el fin de no tomar decisiones equivocadas.

Uno de los aspectos más importantes de un robot autónomo es la navegación. La navegación consiste en poder moverse por el suelo, el aire o el agua, como si lo manejase una persona. Dependiendo del entorno para el cual los robots autónomos han sido diseñados, pueden clasificarse en dos grandes categorías: UAV (Unmanned Aerial Vehicle) y UGV (Unmanned Ground Vehicle). Los UAV [2] son vehículos aéreos no tripulados, generalmente utilizados en labores de reconocimiento y en entornos militares. A su vez, los UGV [2] son aquellos vehículos no tripulados que se mueven por

tierra, como pueden ser los robots utilizados en desactivación de explosivos o de ayuda en rescates.

Los robots autónomos pertenecientes a cualquiera de las dos categorías citadas con anterioridad, UAV y UGV, hacen uso de los datos recogidos por los sensores que poseen, utilizándolos como feedback para mantenerse estables y poder realizar diversas acciones.

Los siguientes apartados recogen brevemente los aspectos más importantes de los UAV y los UGV. También se exponen los principales rasgos que caracterizan a los sensores más relevantes que se utilizan en este tipo de robots, así como los conceptos utilizados en la programación de los mismos.

2.1 VEHÍCULOS NO TRIPULADO (UAV Y UGV)

Como bien se ha comentado en el punto anterior, dentro del campo de los robots autónomos se encuentran dos grandes grupos: UAV y UGV. Ambos tipos de robots se caracterizan por los mismos aspectos (salvando las pequeñas diferencias), poseen múltiples sensores y programas de control avanzados y se diferencian en el entorno en el cual son utilizados, los primeros en el aire y los segundos en el suelo.

Un vehículo aéreo no tripulado (por sus siglas del inglés UAV, Unmanned Aerial Vehicle), es un vehículo aéreo autónomo, capaz de volar sin necesidad de piloto humano, gracias a un sistema de pilotaje autónomo.



Fig. 1. Ejemplos de UAV reales del ejercito de EE.UU.

El término no tripulado puede parecer confuso, debido a que su traducción no es del todo exacta: no tripulado se utiliza en este caso como traducción de Unmanned, cuya traducción más ajustada para este caso sería “no pilotado”. Se le denomina así (UAV) por los militares de los EE. UU., ya que es el nombre que dieron a las últimas generaciones de aeronaves capaces de volar sin piloto a bordo.

Tomado literalmente, el término podría describir un amplio rango de dispositivos capaces de operar en el espacio aéreo que va desde una cometa hasta algo más que un avión radio controlado, pasando por los misiles. Estas aeronaves poseen sistemas que combinan información procedente de sistemas de posicionamiento como GPS, navegación mediante GIS, servomecanismos, etc. La CPU que llevan a bordo se encarga de pilotar sin que sea necesario disponer de un humano a bordo.

Hoy en día el país que más aplicaciones y mayor número de ellos tiene operativos es EE.UU., es de suponer que a medida que la potencia de los sistemas de abordo vaya en aumento, las funciones que realizarán estos robots también crezca. El uso de los UAV hoy en día se centra en misiones de reconocimiento y vigilancia.

En lo referente al entorno terrestre, los vehículos no tripulados o UGV (Unmanned Ground Vehicle) consisten en máquinas robóticas con cierto grado de autonomía, capaces de moverse y realizar acciones sin la necesidad de un piloto humano. Así pues, un coche debidamente modificado, puede convertirse en un UGV y moverse por sí solo, sin la necesidad de un conductor.



Fig. 2. Ejemplo de UGV reales.

Los UGV se utilizan sobre todo en labores de reconocimiento, salvamento o desactivación de explosivos, aunque el uso más extendido de estas máquinas se concentra en el ámbito militar.

Los vehículos no tripulados basan su funcionamiento en la navegación [3], que es la parte encargada de controlar el movimiento del robot y de mantener el rumbo y la orientación. Por supuesto que los vehículos autónomos son capaces de realizar diversas acciones o tareas, pero en el caso de los UAV y UGV la navegación es crucial, ya que sin ella no podrían moverse correctamente. Para realizar la navegación, estos robots incorporan varios dispositivos sensores, mediante los cuales pueden conocer su posición en todo momento. Los dispositivos utilizados corresponden a sensores inerciales (INS, IMU, etc.) y a sistemas de posicionamiento como el GPS. Generalmente los robots autónomos no tripulados suelen integrar ambos tipos de dispositivos para conseguir mejores resultados, requiriendo en estos casos el uso de sistemas de fusión de datos de sensores.

Los vehículos no tripulados disponen de una computadora capaz de gestionar los datos provenientes de los distintos sensores, con el fin de realizar los cálculos necesarios y actuar en consecuencia. La CPU de estos robots se encarga de ejecutar el programa de control y simular el comportamiento pre programado, pudiendo llegar a ser un comportamiento autónomo. Estos programas de control pueden ser muy sencillos o enormemente complejos, incluyendo algoritmos de inteligencia artificial de alto nivel para la realización de diversas tareas.

2.2 SENSORES INERCIALES Y DE POSICIONAMIENTO

Los sensores son componentes esenciales para los robots autónomos, ya que de los datos recogidos por éstos, dependen las acciones a realizar. Los sensores de posicionamiento, como el GPS, y los dispositivos de medición inercial, como los acelerómetros o los giróscopos, son los principales sensores que los robots autónomos han de poseer para realizar una correcta navegación (moverse sin desestabilizarse y manteniendo el rumbo fijado) y saber en todo momento en qué posición se encuentran.

Cada uno de estos dos tipos de sensores tiene sus ventajas y sus inconvenientes, por lo que es muy común utilizarlos de manera conjunta, aprovechando así lo mejor de cada uno.

2.2.1 SENSORES INERCIALES

Cuando se trabaja con sensores de medición inercial (cuyas siglas en inglés corresponden a IMU o INS) [7], se utilizan acelerómetros y giróscopos, los cuales permiten medir los movimientos que realiza el elemento sobre el cual se encuentran instalados, que en el caso de este proyecto es un robot. Los sensores inerciales trabajan con posiciones locales relativas al vehículo sobre el cual se encuentran

instalados, por lo que no se puede obtener la posición global sin realizar una conversión o integración matemática. Los acelerómetros tienen la capacidad de medir la aceleración lineal que sufre el robot en cada uno de los ejes de coordenadas XYZ (ejes locales relativos al robot), de forma periódica, generalmente unas pocas veces por segundo. Por ejemplo, si el robot empieza a moverse hacia delante y en línea recta, el acelerómetro registrará un aumento de la aceleración en el eje correspondiente, hasta que la velocidad se estabilice.



Fig. 3. Sensor de medición inercial o IMU.

Los giróscopos por otra parte, miden la velocidad angular del elemento sobre el cual se encuentran instalados. En el caso de este proyecto, el uso de un giróscopo permite medir la velocidad con la que el robot realiza los movimientos giratorios sobre cualquiera de los tres ejes XYZ, en grados por segundo.

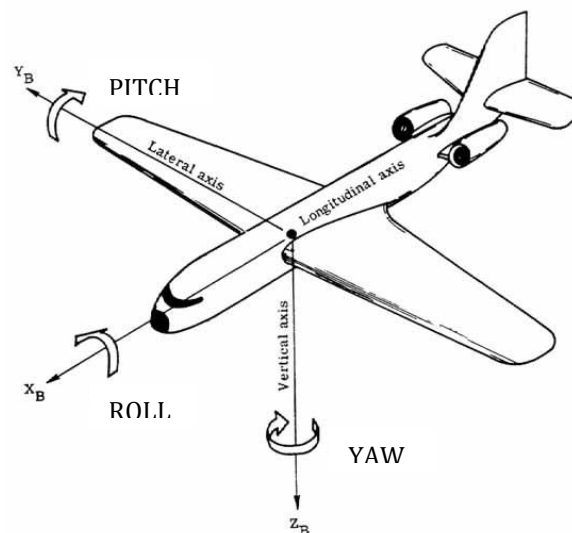


Fig. 4. Componentes de un sensor de medición inercial.

Generalmente, los sensores inerciales de grandes prestaciones, además de un acelerómetro y un giróscopo, poseen un magnetómetro capaz de medir el componente magnético en cada una de las direcciones apuntadas por los ejes de coordenadas XYZ.

En cuanto al uso de estos dispositivos, los datos medidos por los acelerómetros y los giróscopos, son datos relativos a la posición inicial del elemento sobre el que se encuentren instalados, en este caso un robot; por lo que no se pueden utilizar directamente los datos medidos por estos sensores para conocer la posición en la que se encuentra el robot. Es necesario realizar múltiples cálculos matemáticos para obtener las coordenadas de la posición. A estos cálculos se les llama integración (integración matemática) [8], ya que en fondo se trata de integrar la aceleración hasta obtener posición y de integrar velocidad de giro hasta obtener la orientación o el ángulo de giro.

Si bien es cierto que estos dispositivos de medición inercial poseen una precisión bastante elevada, del orden de centímetros, por lo que su uso en tareas de navegación es altamente recomendable, es necesario realizar muchos cálculos por segundo si se desea obtener la posición, lo que puede limitar bastante su uso en equipos de pocas prestaciones. Las principales desventajas son el coste de los dispositivos de este tipo en términos de altas prestaciones, que pueden ser muy elevados en comparación con sistemas de posicionamiento global o GPS, y el problema de la deriva, causado por la acumulación de los pequeños errores en los cálculos de la posición, lo que con el paso del tiempo supone una desviación notable con respecto a la posición real [9]. Por este motivo, los sensores inerciales se suelen utilizar en conjunción con los sistemas GPS, los cuales no se ven afectados por la deriva, por lo que son aptos para ser utilizados como correctores de la posición obtenida por los INS.

En la actualidad, la alternativa más utilizada es el uso de sistemas inerciales de bajo coste, similar al de los sistemas GPS, los cuales requieren de correcciones de forma frecuente para evitar el efecto de la deriva en la medida de lo posible.

2.2.2 SENSORES DE POSICIONAMIENTO

Los sistemas de posicionamiento global (GPS) [10], muy conocidos por la mayoría de las personas debido a su creciente uso en el campo del automóvil e incluso en el de la telefonía móvil, se utilizan para medir la posición del elemento en el cual se encuentran instalados, un robot en el caso de este proyecto. La posición que mide este tipo de dispositivos, es una posición relativa a todo el globo terrestre, y por tanto se utilizan los valores de longitud y latitud para indicar la posición exacta. Además proporcionan información sobre la altitud.

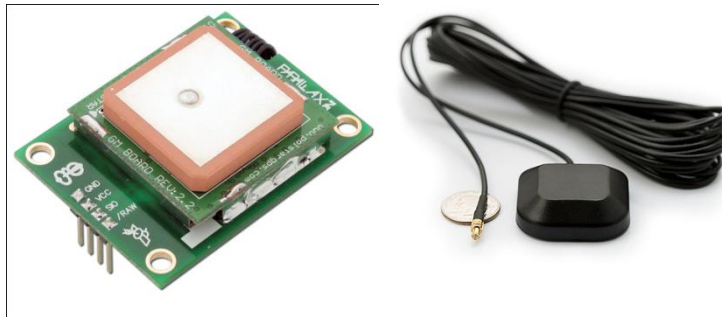


Fig. 5. Sensor y antena GPS.

Para obtener la posición, los dispositivos GPS se conectan como mínimo a cuatro de los más de 20 satélites que orbitan sobre el globo terrestre, con el fin de obtener los datos necesarios que permitan calcular la posición actual del elemento sobre el cual se encuentran montados, en este caso un robot. Los satélites envían al receptor GPS la información de su posición y de su reloj (el tiempo en segundos), de forma que el receptor GPS sincroniza su reloj con la información recibida por cada uno de los satélites, para obtener el retardo de la señal, la cual determina la distancia entre el receptor y el satélite. Con las distancias obtenidas a partir de cada uno de los satélites, el receptor GPS calcula la posición en la que se encuentra a través de una sencilla triangulación.

Los dispositivos GPS tienen dos grandes inconvenientes. Uno de ellos es la precisión con la que miden la posición. Esta precisión suele ser de pocos metros, lo que indica un margen de error muy elevado y para nada aceptable si se utiliza para la navegación de robots autónomos. Además, la tasa de actualización de estos sensores es casi siempre menor que la de los sensores inerciales, del orden de uno o dos datos por segundo. El otro gran inconveniente es que es prácticamente inutilizable en interiores, debido a que la señal de los satélites no puede atravesar las paredes de los edificios o construcciones.

Los dispositivos GPS son muy útiles para localizar la posición global en cualquier momento, de forma poco precisa pero fiable. En cambio, los sensores inerciales son muy precisos pero no indican la posición global, miden las variaciones en el movimiento y por tanto necesitan de cálculos matemáticos y de un punto de referencia para obtener la posición.

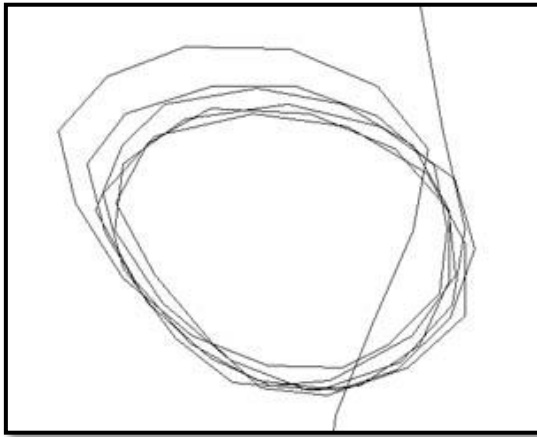


Fig. 6. Trayectoria circular medida por un GPS.

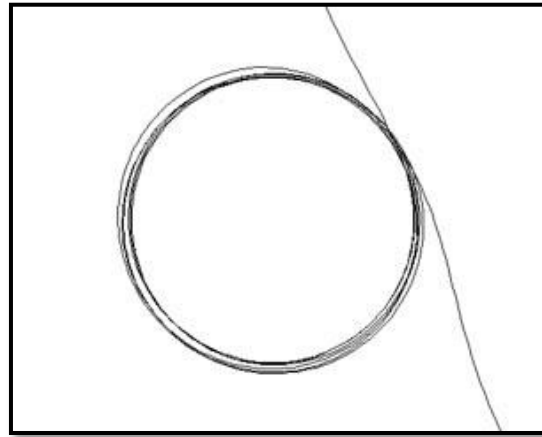


Fig. 7. Trayectoria circular medida por un giróscopo.

En sistemas avanzados, es común utilizar ambos dispositivos y obtener así lo mejor de cada uno de ellos, por un lado la precisión de los sensores inerciales (puede verse una comparativa de la precisión entre una IMU y un GPS en las figuras 6 y 7) y por otro lado la facilidad de medir la posición de los GPS, posición que a pesar de no ser muy precisa, puede utilizarse como punto de referencia.

Combinar los dos tipos de dispositivos de medición es lo más recomendable, pero no es una tarea sencilla de realizar. Para ello, se aplican los denominados algoritmos de fusión de datos, que como su propio nombre indica, combinan los datos de diferentes sensores para mejorar la precisión y la fiabilidad.

2.3 SISTEMAS DE FUSION DE DATOS DE SENSORES

Generalmente, los datos obtenidos por un sensor poseen un cierto error cuantificable. Mediante las técnicas de fusión de datos, es posible incrementar la precisión de la posición obtenida a partir de los datos de los sensores, gracias al uso conjunto de distintos sensores al mismo tiempo [4].

Los robots autónomos, sobre todo los UAV, necesitan que la parte de la navegación sea lo más fiable y exacta posible, ya que de ello depende el buen funcionamiento de todo el robot.

La fusión de datos de múltiples fuentes es un compendio de técnicas multidisciplinarias, análogas al proceso cognitivo que realizamos los humanos, para integrar los datos de múltiples sensores (sentidos) con el fin de realizar inferencias sobre el mundo exterior, convergiendo en un conjunto de resultados (reacción).

Así la fusión de datos pretende obtener un resultado de mejor calidad, a partir de múltiples sensores, eventualmente heterogéneos, realizando inferencias que pueden no ser posibles a partir de uno solo. Teniendo múltiples aplicaciones tanto en el

mundo militar, reconstitución de imágenes, diagnosis médica y en la última década en el mundo del transporte.

2.3.1 NIVELES DE FUSIÓN

La Joint Directors of Laboratories (JDL), que constituye el primer comité científico técnico encargado de estandarizar los conceptos y técnicas de esa disciplina, concibe la fusión como un proceso integral de tratamiento de datos subdividido en niveles que van desde la captura de los datos hasta el resultado final, incluyendo la interacción de dichos resultados con el receptor y/u operador.

Nivel 0: Preproceso de las fuentes

Los datos de las fuentes son habitualmente preprocesados para una fusión posterior, filtrando datos o alineándolos temporalmente u otras acciones previas.

Este proceso lo lleva a cabo, por lo general, el software asociado a cada uno de los sensores comerciales de forma independiente del resto de sensores. Se puede considerar así una prefusión.

Es a este nivel donde cada uno de los sensores aporta sus datos obtenidos de forma independiente del resto.

Nivel 1: Evaluación del objeto

Combinación de los datos de los sensores para obtener posición, velocidad, atributos y características de la entidad. Siendo la entidad, en transporte, por ejemplo un vehículo, un incidente de tráfico o la congestión.

Los algoritmos de este nivel son:

- ⌘ Alineación de datos: Ajustes espacio-tiempo y de unidades para posibilitar un procesamiento posterior
- ⌘ Correlación Dato/Objeto: Asociación y correlación de datos para cada individuo. Para permitir la correcta agrupación de los datos
- ⌘ Estimación de posición, cinemática y otros atributos del objeto: Mediante la combinación de modelos físicos y supuestos estadísticos para la obtención del Vector de Estado.
- ⌘ Estimación de la identidad del objeto: Obtención de la entidad mediante métodos paramétricos (Bayes, Dempster-Shafer), no paramétricos (redes neuronales) o de lógica difusa.

Nivel 2: Evaluación de la situación

Interpretación de los resultados del nivel anterior. Si se determina en el paso anterior que las velocidades son bajas podemos interpretar que estamos en un estado de congestión.

Las técnicas más adecuadas para este nivel son la Inteligencia artificial (IA) y el razonamiento automático.

Los algoritmos de este nivel son:

- ⌘ Agregación de objetos: Agregar los datos de cada una de las identidades para una visión global de la situación.
- ⌘ Interpretación contextual: Tener en cuenta los factores externos al estudio que pueden afectar de forma indirecta, como factores climatológicos, etc.
- ⌘ Evaluación multi perspectiva: Observación global del problema, desde el exterior e interior.

Nivel 3: Evaluación del futuro

Proyección de futuro de la entidad analizada a partir de la situación actual. Los modelos utilizados para este nivel son la inteligencia artificial, los modelos predictivos, el razonamiento automático y la estimación estadística.

Los algoritmos de este nivel son:

- ⌘ Estimación/Agregación de capacidad de fuerza: Agregación de la información de diversos subsistemas para determinar sus interrelaciones y la robustez del sistema.
- ⌘ Estimación de implicaciones: Resultados de una hipotética acción sobre el sistema
- ⌘ Evaluación multi perspectiva: Análoga al del nivel 2, observación del sistema desde el exterior e interior.

Nivel 4: Proceso refinamiento

Meta proceso que monitoriza todo el proceso de fusión de datos para mejorar el rendimiento del proceso en tiempo real, mediante la optimización de la utilización de recursos, la modelización sensorial y la computación de medidas de perfeccionamiento.

Este nivel se considera parcialmente dentro-fuera del proceso de fusión, ya que este refinamiento es necesario para el correcto funcionamiento de la fusión y de las operaciones a la que este destinada la fusión de datos.

Los algoritmos de este nivel son:

- ⌘ Gestión de la misión: Dirección de los recursos existentes con el fin de obtener los resultados deseados.
- ⌘ Predicción de Entidad: Definir que entidades específicas debe reconocer el proceso.
- ⌘ Requerimientos de las fuentes: Definir la infraestructura necesaria de las fuentes para que puedan identificar las entidades
- ⌘ Modelización del rendimiento del sistema: Definir la estructura del sistema de fusión de datos, la relación entre fuentes, los componentes de procesamiento, etc.
- ⌘ Control del sistema: Tal como control de multi objetivos y optimización.

Nivel 5: Refinamiento cognitivo

Mejora del sistema de fusión a partir de la relación Sensor-Procesamiento-Persona, proyectando modelos en los que la representación de los resultados (ya sean intermedios o finales) permita al operador identificar posibles errores en el procesamiento, y que este previamente haya podido detectar errores en los datos suministrados por los sensores.

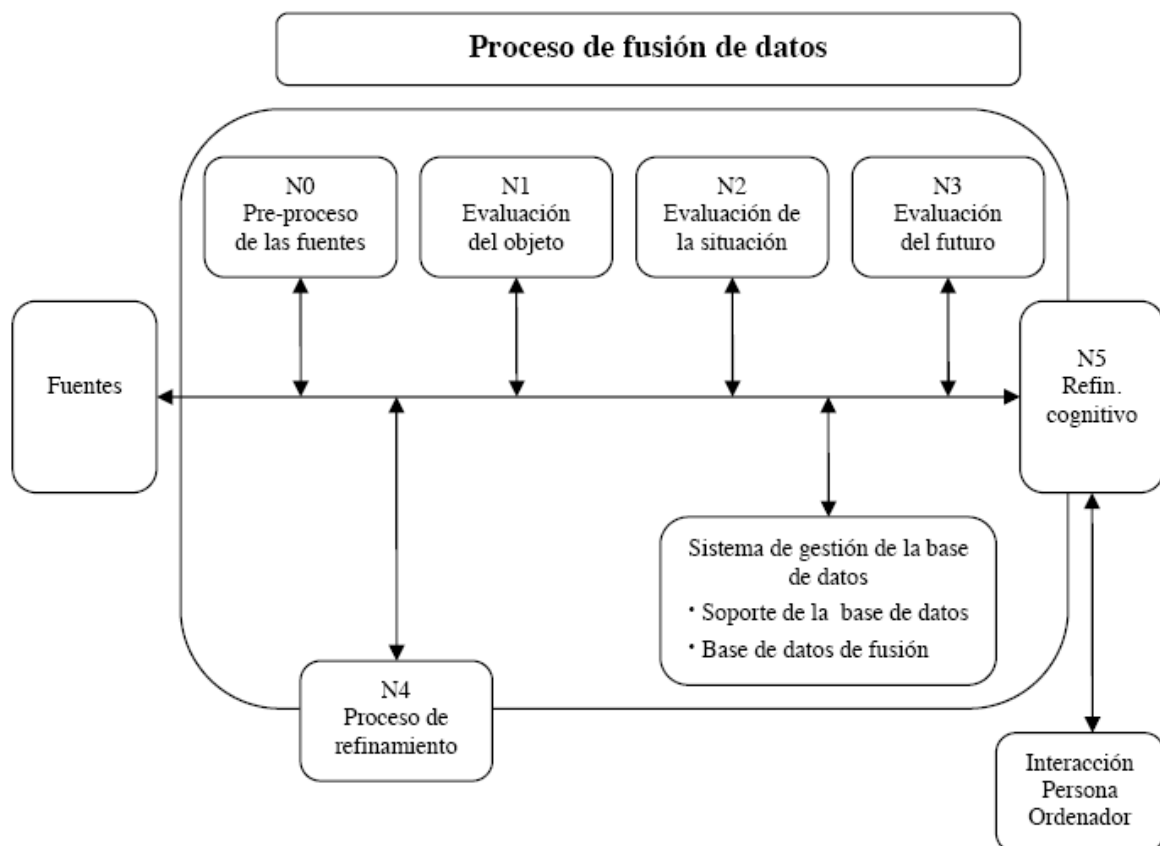


Fig. 8. Esquema del proceso de fusión de datos.

2.4 SISTEMAS DE CONTROL DE ROBOTS

Los robots son controlados generalmente por programas que previamente han sido implementados en algún tipo de lenguaje de programación por el desarrollador. Estos programas dicen al robot lo que tiene que hacer en todo momento, como moverse hacia delante o detenerse porque hay un obstáculo. Desde los inicios de la robótica, estos programas siempre han estado limitados a lo que el diseñador haya especificado en ellos, por lo el robot siempre ejecuta las mismas acciones, no es capaz de resolver nuevas situaciones por sí solo.

Para aumentar el grado de autonomía de los robots e intentar que estos lleguen a simular diferentes comportamientos, cada vez existen más dispositivos sensoriales capaces de medir diferentes aspectos del entorno del robot, como por ejemplo la distancia hasta un obstáculo, la temperatura del ambiente, o la posición global por citar algunos de ellos.

Dejando a un lado las capacidades de los robots en cuanto a autonomía y comportamiento se refiere, todo sistema de control de un robot se basa en los mismos conceptos, los cuales permiten construir sistemas muy sencillos o sistemas realmente complejos y avanzados. En este apartado se describe brevemente los conceptos básicos por los que se rige cualquier sistema de control de robots, con el fin de comprender mejor alguno de los objetivos de este proyecto.

2.4.1 SISTEMAS DE LAZO ABIERTO

Los sistemas de control de lazo abierto [11], son aquellos sistemas en los que no existe una retroalimentación, es decir, que no se utiliza la información de salida para reajustar la entrada de los datos al controlador. Un ejemplo del uso de este tipo de sistemas, sería por ejemplo si el programa de control estuviera diseñado para hacer que el robot se mueva hacia delante continuamente, sin comprobar nada.

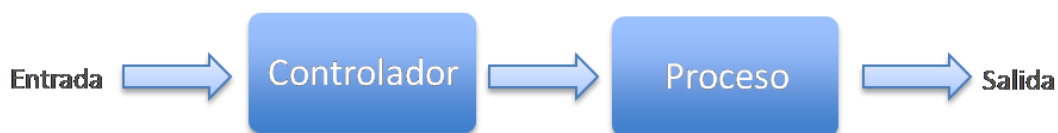


Fig. 9. Esquema de los sistemas de control de lazo abierto.

2.4.2 SISTEMAS DE LAZO CERRADO

Los sistemas de control de lazo cerrado [12], son aquellos sistemas en los que existe una retroalimentación, es decir, que los datos obtenidos a la salida del controlador se utilizan de nuevo para realizar ajustes en las siguientes iteraciones del programa de control. Esto resulta muy útil cuando se necesita conocer el resultado de ciertas acciones para realizar las siguientes. Estos sistemas se utilizan con los robots sensorizados, ya que se utiliza la información de los sensores para reajustar ciertos aspectos en el comportamiento del robot.

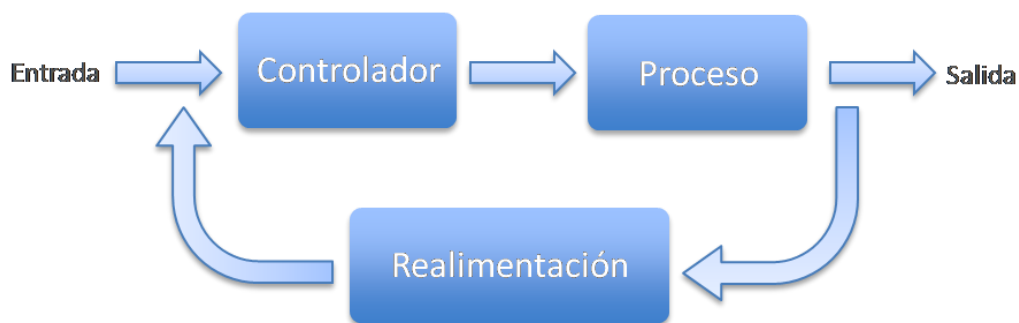


Fig. 10. Esquema de los sistemas de control de lazo cerrado.

En este proyecto se implementa un sistema de lazo cerrado, ya que se utilizan los dispositivos sensores para ajustar las entradas del sistema y tomar las decisiones correctas en cada momento.

EL ROBOT GUARDIÁN

CAPITULO

En la realización de este proyecto se ha utilizado un robot real, el robot Guardián, proporcionado por Robotnik Automation S.L. Se trata de un robot terrestre similar a un vehículo todoterreno, el cual puede desplazarse por diversas superficies gracias a sus cuatro ruedas, e incluso puede llegar a subir y bajar pequeños desniveles o escalones, haciendo uso de una cinta dentada diseñada especialmente para este propósito. Posee un gran número de dispositivos sensores, como GPS, medidor de distancias láser, sensor inercial, etc.; que ofrecen un amplio abanico de posibilidades. Además, alberga en su interior un pequeño PC, destinado a ser el principal sistema de control del robot y mediante el cual es posible ejecutar programas de control y manipular cada uno de los dispositivos. En definitiva se trata de un robot muy completo y apto para utilizarse como un UGV (robot terrestre no tripulado) [2].

En este capítulo se recogen los aspectos más significativos del robot Guardián, detallando sus características y su funcionamiento.



Fig. 11. El robot Guardián.

3.1 CARACTERÍSTICAS DEL ROBOT

El robot Guardián es un vehículo terrestre con una alta movilidad sobre diversas superficies y entornos. Sus principales especificaciones técnicas son las siguientes:

- ⌘ Dimensiones: 1050 x 600 x 400 mm.
- ⌘ Peso: 75 Kg.
- ⌘ Capacidad de carga: 50 Kg.
- ⌘ Velocidad: 1.25 m/s.
- ⌘ Arquitectura software basada en Player/Stage.
- ⌘ Software en código abierto.

El robot Guardián cuenta con diversos dispositivos (o sensores) que permiten conocer en todo momento el estado del robot. Estos dispositivos recogen datos periódicamente sobre distintos aspectos, como puede ser la posición satelital o la cercanía de objetos. Los datos recogidos por los sensores, pueden ser utilizados por el robot para realizar una navegación segura, evitando obstáculos [18] o manteniendo el rumbo deseado, o para llevar a cabo ciertas acciones con precisión. Los distintos dispositivos que posee el robot Guardián son los siguientes:

- ⌘ Dispositivo de posicionamiento global (GPS).
- ⌘ Dispositivo de medición inercial compuesto por acelerómetro, giróscopo y magnetómetro (IMU).
- ⌘ Dispositivo de medición láser.
- ⌘ Odómetro.
- ⌘ Sensor de infrarrojos para la detección de líneas.
- ⌘ Sensores de infrarrojos de proximidad (en la parte trasera del robot).
- ⌘ Cámara IP a color.

A continuación se muestra una imagen esquemática del citado robot, en la que se puede apreciar las distintas partes que lo componen y sus ubicaciones en la estructura del robot.

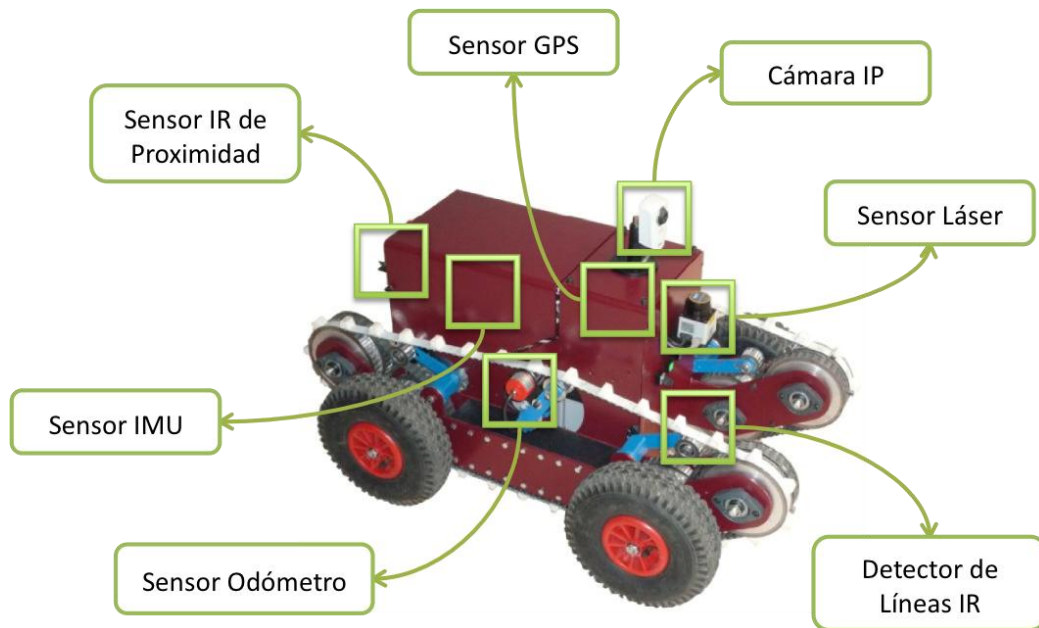


Fig. 12. Diferenciación de las distintas partes del robot Guardián.

Para que todos los dispositivos funcionen correctamente, el robot cuenta con un ordenador personal de reducidas dimensiones, el cual está situado bajo la cubierta superior. Este PC se encarga de controlar todos los dispositivos y de ejecutar los programas de control del robot, por lo que todos los dispositivos se conectan al PC, a excepción de la cámara IP, la cual se conecta directamente al punto de acceso, permitiendo controlar la cámara vía Web. Las características del PC son las siguientes:

⌘ CPU:

- ❖ Placa base: K8-Triton Series GA-K8VT800 (2.0)
- ❖ Procesador: AMD Athlon K8
- ❖ Tarjeta de red: RTL8100C chip (10/100Mbps) - 1 puerto RJ 45
- ❖ Slots:

- 5 PCI slots, 1 AGP 8x/4x (1.5V)

❖ Periféricos:

- 1 puerto paralelo
- 2 puertos serie
- 8 puertos USB 2.0/1.1
- 1 conector audio
- 1 conector IR
- 1 conector PS/2 teclado
- 1 conector PS/2 ratón

⌘ Tarjeta Gráfica: NVIDIA GeForce

- ⌘ Tarjeta CAN Bus: ESD PCI331
- ⌘ Joystick: Logitech Extreme 3D Pro

El sistema operativo del PC es de tipo Linux, concretamente una modificación de la distribución Xubuntu 8.01 adaptada a las necesidades del robot y de su software de control.

3.2 FUNCIONAMIENTO Y CONTROL

El robot Guardián alberga un PC en su interior, el cual es accesible desde el exterior vía WiFi, gracias al punto de acceso al que está conectado. Las ventajas de utilizar un PC, en vez de pequeños microprocesadores, son casi evidentes. Un PC proporciona una mayor potencia de procesamiento y versatilidad, además de permitir el uso de software ya existente. Por el contrario el precio es bastante más elevado que el de un pequeño microprocesador y los demás componentes necesarios, así como el tamaño y el peso del PC son también mucho mayores. Dado que el tamaño no es un impedimento para el robot Guardián y debido a sus múltiples dispositivos sensores, la mejor opción es sin duda la de utilizar un PC.

En lo que respecta a los actuadores del robot Guardián, únicamente se encuentra disponible un motor eléctrico encargado de impulsar el sistema de movimiento del robot, de tipo diferencial y compuesto por un total de cuatro ruedas, dos a cada lado del robot. Así pues, las únicas acciones “físicas” que el robot puede llevar a cabo, se limitan al movimiento por el entorno.

El robot puede ser controlado remotamente, accediendo al PC de forma inalámbrica. El control está basado en el envío de comandos y la recepción de información. Todo se realiza mediante el envío de paquetes a través del protocolo TCP/IP [13]. El control del robot consiste en la ejecución de un programa codificado con anterioridad, de tal modo que dicho programa puede haber sido realizado para permitir al usuario mover el robot mediante una combinación de teclas, o bien para que sea el propio robot el que se mueva por sí solo de acuerdo a las instrucciones preprogramadas. Esto son dos posibles ejemplos de utilización de programas de control y que ilustran las posibilidades del robot al permitir ser controlado por un usuario de forma remota, o por el contrario, ser autónomo y no depender de las decisiones del usuario.

Los programas de control del robot Guardián, se implementan con la ayuda de software externo. Se trata de una plataforma de código abierto orientada a aparatos robóticos, llamada Player/Stage [5] [6].

PLAYER/STAGE

CAPITULO

Los robots son piezas complejas de hardware, compuestos por múltiples mecanismos y dispositivos, los cuales en su conjunto hacen posible que dichas máquinas puedan llegar a realizar acciones e incluso simular comportamientos, que en algunos casos pueden considerarse autónomos. Si bien es cierto que la parte mecánica de los robots es fundamental, sin el correspondiente software de control, los robots no serían más que un conjunto de piezas y engranajes sin utilidad aparente, muy lejos de poder aparentar algún grado de autonomía o inteligencia.

La programación de los robots brinda la posibilidad de controlar las distintas piezas de las que se compone un robot, de forma que estas máquinas puedan realizar multitud de tareas diferentes. Para programar el control de un robot, siempre ha sido necesario utilizar métodos y lenguajes específicos, muy ligados al hardware del robot para el que se pretendía programar, lo que implicaba muy poca flexibilidad y por tanto hacía imposible la estandarización de los aspectos de la programación de robots. Hoy en día la situación está cambiando, gracias a la aparición de plataformas y herramientas que generalizan la programación de los robots y reducen la dependencia de los programas de control con el hardware específico.

Player/Stage es un conjunto de herramientas de código abierto que ofrecen facilidades para el desarrollo de programas de control. La plataforma Player/Stage actúa como un pequeño sistema operativo permitiendo controlar los distintos componentes de un robot a través de un sencillo sistema de interfaces genéricas, independientes del hardware específico del robot, por lo que resulta muy sencillo realizar programas para distintos robots, sin que el tipo de los componentes sea un impedimento.

4.1 NECESIDAD DE UTILIZAR HERRAMIENTAS SOFTWARE

Para la programación de controladores sencillos para pequeños robots, los cuales desempeñan tareas simples, se han utilizado y se utilizan pequeños micro controladores, como los PIC [15]. Los programas simples no requieren grandes capacidades de procesamiento, ni de memoria, por lo que estos pequeños micro controladores ofrecen las prestaciones necesarias para este tipo de programas y robots.

Actualmente, el interés por los robots ha aumentado considerablemente, gracias a las nuevas posibilidades que ofrecen los avances tecnológicos, que permiten dotar a estas máquinas de una mayor capacidad de procesamiento y de almacenamiento, así como de sensores mucho más precisos.

De los pequeños programas para robots, los cuales implementan tareas muy sencillas, se ha pasado a programar tareas mucho más complejas. Estas tareas tienen unos requerimientos mucho mayores que los que pueden cubrir los micro controladores sencillos utilizados hasta ahora. Por este motivo, se ha pasado a utilizar sistemas mucho más avanzados, prácticamente ordenadores personales.

Junto con el uso de PCs para controlar a los robots, se utilizan herramientas de programación de más alto nivel que las empleadas en la programación de micro controladores del tipo PIC. Esto supone ventajas como la abstracción o la reutilización de código, e inconvenientes como el de la eficiencia (en comparación a la microprogramación).

Si bien la tecnología no es uno de los principales problemas para el software actual, el desembolso económico puede ser un fuerte impedimento cuando se pretende adquirir un robot. Este es uno de los motivos por los cuales se utilizan simuladores para probar los programas y comprobar los resultados obtenidos, acercándose lo más posible a los que podrían obtenerse con el robot real.

El uso de un simulador permite ejecutar los programas como si del propio robot se tratase, aunque cabe destacar que las condiciones del entorno simulado no son las mismas que las del entorno real, el cual presenta ruidos en las mediciones de los sensores, situaciones inesperadas, etc., aspectos que a veces no pueden ser reproducidos con fidelidad en la simulación. Los simuladores suelen ser utilizados para comprobar los programas que posteriormente serán implantados en el robot real, de una forma más metódica y exhaustiva, ahorrando en algunos casos bastante tiempo.

Player y Stage son dos herramientas software open source (de código abierto) que ofrecen un completo entorno de simulación si se utilizan de manera conjunta.

4.2 PLAYER

En este apartado se describe de una manera sencilla la herramienta Player, así como se detallan los aspectos más relevantes de la misma.

4.2.1 ¿QUÉ ES PLAYER?

Player es un HAL o Hardware AbstractionLayer (Capa de Abstracción de Hardware) para dispositivos robóticos que funciona bajo sistemas Linux. Se podría decir que es un pequeño sistema operativo de drivers en el cual se pueden manejar diferentes tipos de dispositivos como sensores laser, infrarrojos, GPS, etc. Como principales características de Player se pueden destacar las siguientes:

- ⌘ Permite el acceso a robots y sensores a través de la red.
- ⌘ Incluye drivers desarrollados de dispositivos utilizados en el robot.
- ⌘ Posee librerías para C, C++, Java, Lisp y Python.

Player proporciona una interfaz de comunicación clara y simple con los sensores y los accionamientos del robot, a través de una red TCP/IP [13]. También define una serie de interfaces estándar, cada una de las cuales es una especificación de las formas en las que se puede interactuar con alguna clase de dispositivos. Estas interfaces proporcionan al desarrollador funciones de alto nivel para el acceso y configuración de los diferentes dispositivos o sensores.

4.2.2 PLAYER Y LA ARQUITECTURA CLIENTE-SERVIDOR

Player es el módulo que define la arquitectura del sistema. Se puede descomponer en dos partes claramente diferenciadas:

- ⌘ Por un lado hay un entorno servidor que se ejecuta sobre el robot (real o simulado) que recibe peticiones TCP sobre el protocolo IP.
- ⌘ Por otro lado hay un cliente que accede a ese robot, para lo que existen unas librerías cliente que evitan trabajar a bajo nivel. El programa cliente se comunica con el robot a través de sockets TCP [14], permitiendo la lectura de los sensores, accionar los mecanismos del robot, así como la configuración de ciertos parámetros y dispositivos del robot.

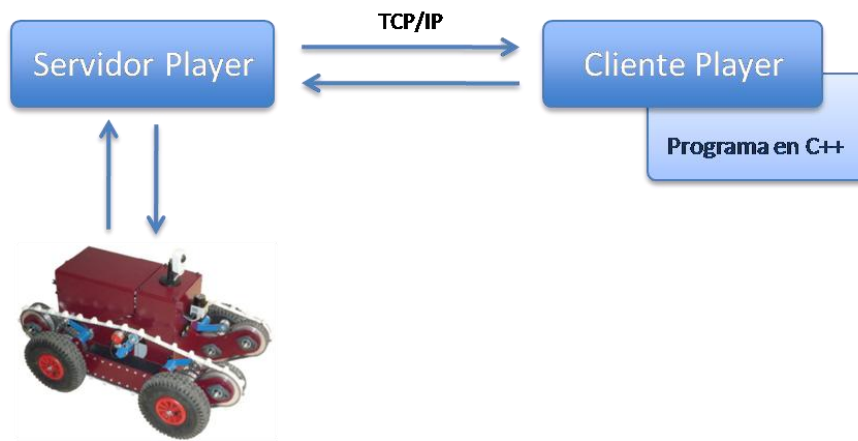


Fig. 13. Arquitectura cliente-servidor de Player.

El desarrollador sólo necesita escribir un programa cliente que haga uso de las interfaces proporcionadas por las librerías de Player. Las librerías se encargan del trabajo sucio de contactar con el servidor y enviar mensajes, de forma que sólo se necesita usar dichas interfaces para controlar el robot. Con esta arquitectura, un mismo cliente que use unas interfaces concretas, puede controlar diferentes robots que tengan los drivers apropiados para implementar las mismas interfaces (incluso aunque el hardware sea muy diferente). En definitiva, los drivers comunican el hardware del robot con los programas cliente a través de las interfaces estándar. De este modo el hardware del robot es transparente para el desarrollo de programas.

4.2.3 INTERFACES, DRIVERS Y DISPOSITIVOS EN PLAYER

Antes que nada, es necesario entender que significan los términos utilizados hasta ahora. A continuación se da una breve definición de algunos términos importantes:

- ⌘ **Interfaz:** Una interfaz ofrece un conjunto de operaciones para acceder y controlar un sensor. Permite controlar un dispositivo a alto nivel y de forma genérica, dejando de lado el lenguaje específico del propio hardware del dispositivo.
- ⌘ **Driver:** Un driver es el software encargado de controlar los dispositivos de forma directa. Se comunica con el hardware en su formato específico y a su vez implementa una o varias interfaces, para que “otros” puedan acceder al dispositivo de forma genérica, abstrayéndose del lenguaje específico del hardware.
- ⌘ **Dispositivo (Sensor):** Es la pieza hardware en sí, un sensor o un actuador.

Estos elementos están relacionados entre sí, de forma que un programa cualquiera utiliza las interfaces para enviar comandos a los sensores del robot. A su

vez, las interfaces se comunican con los drivers y son éstos los que se comunican de forma directa con los dispositivos.

Como ya se ha visto, Player posee dos partes bien diferenciadas: un servidor que se ejecuta en el robot y se comunica con su hardware y un programa cliente que se encarga de enviar peticiones al servidor. Esta comunicación se realiza a través de unas interfaces estándar que proporcionan las librerías de Player. Dichas interfaces proveen al desarrollador funciones de alto nivel para comunicarse con el servidor que se está ejecutando en el robot, evitando la parte engorrosa de la comunicación entre un servidor y un cliente mediante sockets TCP. Del mismo modo, el servidor implementa las mismas interfaces que el cliente, de forma que la comunicación sea posible.

A parte de las interfaces para la comunicación entre cliente y servidor, existen interfaces para comunicarse con los dispositivos del robot (láser, motor, GPS, etc.). Gracias a estas interfaces, el hardware específico de cada robot (el tipo de sensores y actuadores) es indiferente para el desarrollo de los programas (clientes).

Esta abstracción se consigue gracias al uso de drivers (controladores), que son los encargados de comunicarse directamente con los dispositivos. Los drivers sí son específicos para cada tipo de hardware. Por ejemplo, para comunicarse con un dispositivo láser Hokuyo, se utiliza el driver “urglaser”. Este driver no serviría si hubiera que comunicarse con un medidor láser SICK LMS-200, para lo cual se utilizaría el driver “sicklms200”. (Ambos drivers están implementados en Player por defecto). Lo que tienen en común ambos drivers, es que implementan la interfaz estándar “laser”, a través de la cual se pueden enviar comandos para obtener datos de los sensores láser. Gracias a que ambos drivers implementan la misma interfaz, un programa que utilice esta interfaz para comunicarse con un sensor láser, podrá ser válido para ambos dispositivos, sin importar que el hardware sea diferente. Únicamente cambiará el driver a utilizar.

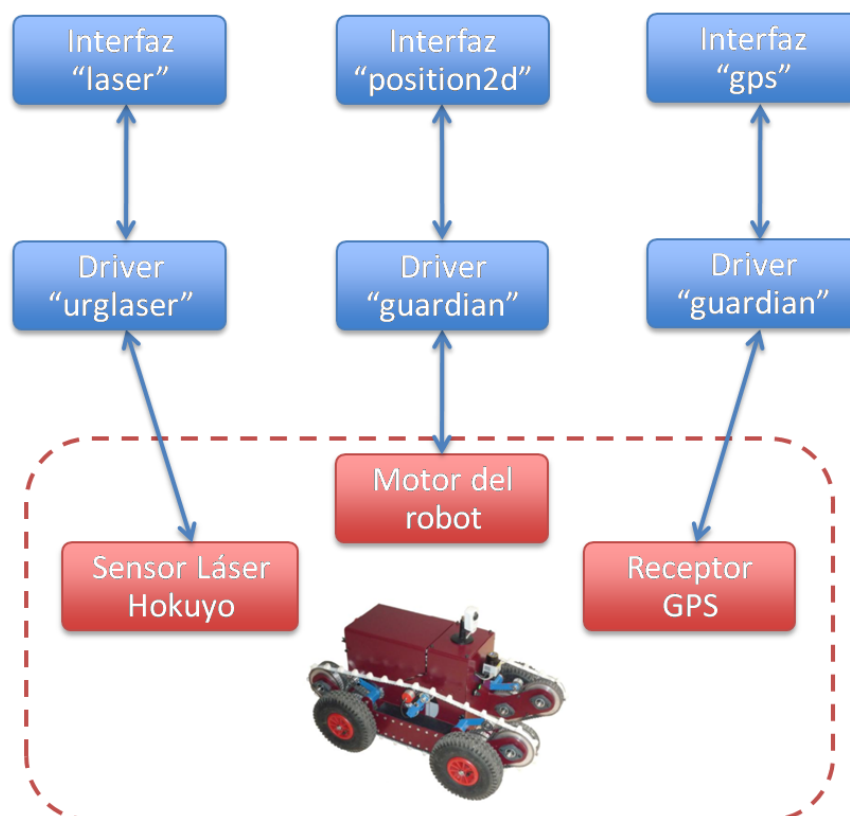


Fig. 14. Esquema de acceso al robot "interfaces/drivers/dispositivos".

Cabe destacar que un driver puede implementar varias interfaces y no necesariamente una sola. Este es el caso del robot Guardián, cuyo driver "guardian" implementa las interfaces "position2d", "imu", "gps", "aio", "dio", "power" y "opaque", a través de las cuales puede controlarse el robot. El medidor de distancias láser del robot Guardián no se puede controlar con el driver "guardian", pero si es accesible a través del driver "urglaser", el cual viene implementado en Player por defecto y ofrece una interfaz llamada "laser" para el acceso al dispositivo láser.

En la figura 13 puede verse un ejemplo de la estructura de la comunicación con el robot. La figura ilustra un robot, el cual posee un sensor láser Hokuyo, un motor y un receptor GPS. La comunicación directa con estos dispositivos o sensores, se realiza a través de los drivers. En este ejemplo, el driver "guardian" se utiliza para comunicarse con dos dispositivos a la vez, mientras que para la comunicación con el sensor láser, se utiliza el driver "urglaser". A su vez, para la comunicación con los drivers se utilizan unas interfaces, las cuales ofrecen una serie de funciones para la comunicación a más alto nivel.

Con el ejemplo anterior, es fácil observar que un programa que utilice las mismas interfaces para comunicarse con un robot (X), puede utilizarse para controlar otro robot (Y) cuyo hardware sea igual o distinto. Simplemente hay que sustituir los drivers necesarios.

4.2.4 FICHEROS DE CONFIGURACIÓN DEL SERVIDOR PLAYER

Características generales

Llegados a este punto, está claro que un programa cliente se comunica con los dispositivos de un robot a través de interfaces estándar, válidas para diferentes dispositivos del mismo tipo, siempre y cuando se disponga del driver apropiado. Pero para que esto sea posible, es necesario vincular de alguna forma los dispositivos con los drivers y con las interfaces.

En Player, la asociación de interfaces/drivers/dispositivos se realiza en el servidor, que es el que se estará ejecutando en el propio robot. Para ello, se debe de configurar un fichero, normalmente con extensión “.cfg”, en el que se especifican las asociaciones entre drivers e interfaces. Concretamente, se indican los drivers que van a utilizarse y las interfaces que cada uno de éstos drivers utiliza. Asimismo, se indica el dispositivo vinculado a ese driver y esa interfaz.

A continuación se muestra un ejemplo sencillo de un posible fichero de configuración de Player. En este ejemplo se pretende asociar un sensor láser SICK LMS-200 con la interfaz estándar “laser”, todo ello a través del driver específico para el sensor láser SICK LMS-200.

```
driver
(
    name "sicklms200"
    provides ["laser:0"]
    port "/dev/ttyACM0"
)
```

Fig. 15. Ejemplo de un fichero de configuración de Player.

Como puede observarse en el ejemplo, en un fichero de configuración se especifican bloques de drivers. Dentro de cada bloque, el cual corresponde a la definición de un driver específico, se indican varias cosas:

- ⌘ El nombre del driver (necesario para que Player cargue el driver en la ejecución).
- ⌘ Los dispositivos que el driver ofrece. Para cada dispositivo, se indica la interfaz que va a utilizarse para comunicarse con él, y el índice del dispositivo (ya que puede haber más de un sensor del mismo tipo). En el ejemplo, puede verse que el driver “sicklms200” ofrece un sensor de tipo “laser” (se indica la interfaz que se va a utilizar para acceder al propio láser) cuyo índice es el “0” (el índice sirve para diferenciar varios dispositivos iguales entre sí).
- ⌘ El puerto a través del cual el driver se va a comunicar con los dispositivos.
- ⌘ Otras opciones más avanzadas.

Estructura y sintaxis

La estructura de un fichero de configuración de Player, define una sección por cada driver que se vaya a utilizar. Para cada sección hay que especificar las características del driver y los dispositivos que va a controlar.

Los ficheros de configuración de Player poseen una sintaxis clara y definida, que permite definir los diferentes aspectos de los drivers y los dispositivos que éstos controlan. Cada sección viene definida por la palabra clave “driver”, seguido de unos paréntesis. Dentro de los paréntesis se definen las opciones de cada driver mediante pares de “opción-valor”, los cuales se separan por espacios. Cada driver tiene unas opciones de configuración concretas, aunque existen ciertas opciones independientes de cualquier driver. A continuación se explican las opciones generales, las cuáles son válidas para todos los drivers:

- ⌘ **name** (string): Es el nombre del driver que se va a instanciar. Obligatorio.
- ⌘ **plugin**(string): El nombre de la librería compartida que contiene el driver. Esta librería se intentará cargar antes de instanciar el driver. Se utiliza para drivers no incluidos por defecto en Player.
- ⌘ **provides** (tupla de strings): Los dispositivos a los que se va a poder acceder a través del driver. Como un mismo driver puede ofrecer varios dispositivos, éstos se especifican entre corchetes y separados por un espacio. Obligatorio.
- ⌘ **requires** (tupla de strings): Los dispositivos a los que el driver va a suscribirse, los cuales son accesibles a través de otro driver. El otro driver ha de declararse en primer lugar.
- ⌘ **alwayson** (int): Si es 1, el driver se instancia al arrancar Player, sin esperar a que un programa cliente se conecte.

A parte de estas opciones generales e independientes de los drivers, cada driver puede tener más opciones (para indicar el puerto de conexión, etc.), pero estas opciones son específicas de cada driver y es el desarrollador del driver quien especifica dichas opciones.

Cabe destacar que las opciones de cada driver pueden usar cualquier nombre que el desarrollador del driver quiera darles, con excepción de las siguientes palabras, las cuales están reservadas para su uso general:

- ⌘ driver, name, plug-in, provides, requires, unit_length, unit_angle

Fichero de configuración del robot Guardián

Ya se ha visto que la forma en la que se asocian los sensores con las interfaces en Player, es a través de los ficheros de configuración. A continuación, se expone el caso concreto del fichero de configuración del robot Guardián.

Cabe destacar que la configuración en Player del robot Guardián, es bastante sencilla. Únicamente se utilizan dos drivers para controlar todos los dispositivos.

```
driver
(
    name "urglaser"
    provides ["laser:0"]
    port "/dev/ttyACM0"
)

driver
(
    name "guardian"
    plugin "./bin/libguardian"
    provides ["position2d:0" "power:0" "dio:0" "imu:0" "gps:0"
"opaque:0"]
    serial_port "/dev/ttyS0"
    log_ip "127.0.0.1"
    log_port 15560
    requires ["laser:0"]
)
```

Fig. 16. Fichero de configuración guardian.cfg.

El robot Guardián posee los siguientes sensores o dispositivos, a los cuales se accede a través de las interfaces estándar:

- ⌘ Motor para las ruedas → **position2d**
- ⌘ Sensor láser → **laser**
- ⌘ Sensor GPS → **gps**
- ⌘ Sensor inercial (IMU) → **imu**
- ⌘ Medidor de carga de batería → **power**
- ⌘ Dispositivo de entradas/salidas analógicas → **aio**
- ⌘ Dispositivo de entradas/salidas digitales → **dio**

Todos los dispositivos, menos el sensor láser, son accesibles gracias al driver “guardian”. El sensor láser Hokuyo ya tiene un driver en Player, el “urglaser”, por lo que se utiliza éste último para acceder a este sensor. Como puede verse en la Figura 4, en la definición del driver “guardian”, se requiere el dispositivo “laser:0”, que es proporcionado por el driver “urglaser”. De esta manera, cuando se utilice el driver “guardian”, se podrá acceder a todos los dispositivos a través de las interfaces estándar de Player, incluyendo el sensor láser, sin importar qué driver se comunica con qué dispositivos.

En el fichero de configuración, se indican aspectos como la librería “libguardian”, en la que se encuentran las implementaciones de las interfaces no estándares, como la “imu” o el “power”. También se indican el puerto y las direcciones IP por las cuales se va a acceder a los dispositivos del robot.

El fichero de configuración no es necesario modificarlo, a menos que se desee incluir nuevo hardware en el robot. Para más información sobre los ficheros de configuración de Player, visitar la página Web de Player/Stage sobre configuración [1].

4.2.5 PROGRAMAS CLIENTE EN PLAYER

Para desarrollar un programa para el robot, basta con utilizar las librerías que Player proporciona y hacer uso de las interfaces para controlar el robot. A través de las interfaces, se pueden leer los datos provenientes de los sensores, así como enviar comandos a los mismos.

Los programas pueden desarrollarse en múltiples lenguajes de programación, como C, C++, Java o Python, ya que Player dispone de librerías para todos ellos.

Únicamente es necesario seguir una pauta a la hora de implementar cualquier programa. Tiene que ver con el ciclo de vida del programa y su conexión inicial con el robot.

La estructura de un programa cliente, utilizando las librerías de Player, es la siguiente. Hay que recordar que pueden utilizarse distintos lenguajes de programación, pero en este documento se ilustran los ejemplos en C++:

- ⌘ Conectar el cliente con el robot (con el servidor).
- ⌘ Acceder a los dispositivos o sensores utilizando las interfaces apropiadas. En esta fase se realiza lo que se llama suscripción a las interfaces, que no es otra cosa más que vincular un objeto “proxy” con un dispositivo o sensor. Existe un “proxy” para cada tipo de sensor y son representados por una clase con sus atributos y sus métodos, de forma que trabajar con los sensores es mucho más sencillo y flexible.
- ⌘ Crear un bucle, generalmente infinito, que representa el ciclo de vida del programa.

- ⌘ Leer los datos de los sensores, normalmente una vez por iteración, dependiendo de las velocidades de refresco de los sensores.
- ⌘ Realizar las acciones que se quiera, como hacer girar el robot si hay obstáculos delante.
- ⌘ Cerrar la conexión y salir del programa.

En pseudocódigo, la estructura de un programa Player (explicada arriba) es de la siguiente forma:

```

conectarRobot(IP, puerto)
conectarDispositivos (robot, índice)
while (condicionParada)
{
    leerDatos()
    logicaPrograma()
    enviarComandos()
}
desconectarDispositivos()
desconectarRobot()

```

A continuación se muestra un pequeño ejemplo de un programa cliente que permite esquivar obstáculos gracias a las lecturas del sensor sonar. Mediante este ejemplo, se muestra la estructura de un programa en C++.

```

#include <iostream>
#include <libplayerc++/playerc++.h>

int main(int argc, char *argv[])
{
    using namespace PlayerCc;

    PlayerClient    robot("localhost");
    SonarProxy      sp(&robot, 0);
    Position2dProxy pp(&robot, 0);

    for(;;)
    {
        double turnrate, speed;

        // read from the proxies
        robot.Read();

        // print out sonars for fun
    }
}

```



```

std::cout<< sp <<std::endl;

// do simple collision avoidance
if((sp[0] + sp[1]) < (sp[6] + sp[7]))
    turnrate = dtor(-20); // turn 20 degrees per second
else
    turnrate = dtor(20);

if(sp[3] < 0.500)
    speed = 0;
else
    speed = 0.100;

// command the motors
pp.SetSpeed(speed, turnrate);
}
}

```

Fig. 17. Ejemplo de programa cliente.

En el programa, lo primero es incluir las librerías necesarias que proporcionan las funciones que se van a necesitar. En este caso, se utilizan las librerías de Player para C++, incluidas en la librería padre “libplayerc++”.

```
⌘ include<libplayerc++/playerc++.h>
```

Seguidamente empieza el cuerpo del programa y siguiendo la estructura de todo programa Player, se realiza la conexión con el robot indicando su dirección IP, que en este caso la dirección IP del servidor del robot es la dirección “localhost”.

```
⌘ PlayerClient robot("localhost");
```

Una vez que se realiza la conexión con el robot, se accede a los dispositivos que van a utilizarse, suscribiéndose a ellos a través de un objeto “proxy”, que tal y como se ha mencionado con anterioridad, los “proxys” representan cada uno de los dispositivos, encapsulándolos en un objeto con sus atributos y métodos específicos. En este caso, se dispone de un dispositivo sonar y un motor (accesible a través de la interfaz position2d). Para acceder a ambos dispositivos, es necesario suscribirse a ellos, por lo que se realizan sendas llamadas a los constructores de los “proxys” que representan a cada uno de los dispositivos, indicando el cliente (PlayerClient) con el que se ha establecido la conexión y el índice del dispositivo (por si hay más de un dispositivo del mismo tipo).

```
⌘ SonarProxyp(&robot,0);
```

```
⌘ Position2dProxyp(&robot,0);
```

Después de especificar la conexión y de suscribirse a los dispositivos necesarios, se inicia el bucle principal, que en este caso es infinito, aunque esto depende de lo que se quiera conseguir.

En cada una de las iteraciones se leen los datos de TODOS los dispositivos. Una única llamada al robot, permite que todos los dispositivos vuelquen sus datos sobre los objetos “proxy”. De este modo, después de esta llamada se puede acceder a los objetos vinculados con los dispositivos para obtener sus datos. La llamada de lectura de datos es la siguiente:

```
⌘ robot.Read();
```

La llamada a “Read()” provoca que los datos se actualicen, eliminando los datos anteriores. No es una llamada bloqueante, por lo que si algún sensor no dispone de nuevos datos, no espera por él, sino que vuelca los datos que tenga en ese momento, aunque sean antiguos.

Una vez que se han leído los datos de los sensores, se puede hacer cualquier análisis u operación. En este caso, se analizan los datos provenientes del sensor sonar y si se detecta que el robot tiene un obstáculo delante, se envían instrucciones al motor para que cambie su dirección y velocidad.

La lectura de los sensores depende de la interfaz a utilizar. En este caso, la lectura del sonar es tan sencilla como: `sp[x]`.

El envío de nuevas instrucciones al actuador del motor, se realiza a través de la interfaz “position2d” del siguiente modo:

```
⌘ pp.SetSpeed(speed, turnrate);
```

En definitiva, todos los programas Player siguen la misma estructura y el único aspecto más difícil para el desarrollador, es aprenderse las funciones que proporciona cada una de las interfaces, las cuales permiten el acceso a los dispositivos.

4.2.6 LIBRERÍAS DE PLAYER

Player está compuesto principalmente por cinco librerías, las cuales dan soporte al envío de datos a través de TCP/IP, manejo de errores, manejo de drivers, etc. Estas librerías son las siguientes:

⌘ **libplayererror** (C): Facilita el reporte de errores.

⌘ **libplayercore** (C++): Envío de mensajes, soporte para la carga de plugins lectura de los ficheros de configuración.

- ⌘ **libplayerdrivers** (C++): Los drivers que vienen incluidos en Player por defecto (también vienen compilados).
- ⌘ **libplayertcp** (C++): Soporte para el transporte TCP entre cliente y servidor.
- ⌘ **libplayerxdr** (C++): Soporte para manejo de datos XDR.

Por otro lado, los clientes pueden hacer uso de otras librerías para comunicarse con los dispositivos. Estas librerías están disponibles para varios lenguajes de programación, de forma que el desarrollador tenga más libertad a la hora de programar. Además, ofrecen un completo API que facilita el desarrollo de programas cliente. Las librerías son:

- ⌘ **Libplayerc**: Librería cliente escrita en lenguaje C.
- ⌘ **Libplayerc++**: Librería cliente escrita en lenguaje C++.
- ⌘ **Libplayerc_py**: Librería cliente escrita en Python.

4.3 STAGE

El control de robots mediante la plataforma Player/Stage se realiza únicamente a través de Player y sus librerías. Basta con tener un servidor Player ejecutándose en el robot y un programa cliente que se comunique con el servidor y controle el robot.

Otro punto interesante en la programación de robots es la simulación. Resulta interesante poder simular el comportamiento que un determinado programa tendrá sobre un robot, sin correr riesgo alguno. Mediante la simulación, es posible acercarse bastante a lo que sucede en la realidad. Por eso es muy común probar los programas en un simulador antes de ejecutarlos sobre el robot real. Es aquí donde entra en juego Stage.

4.3.1 ¿QUÉ ES STAGE?

Stage es un entorno completo de simulación para la plataforma Player. Ofrece una interfaz gráfica en dos dimensiones, que imita el mundo real gracias al uso de una imagen que sirve de mapa o plano del entorno del robot.

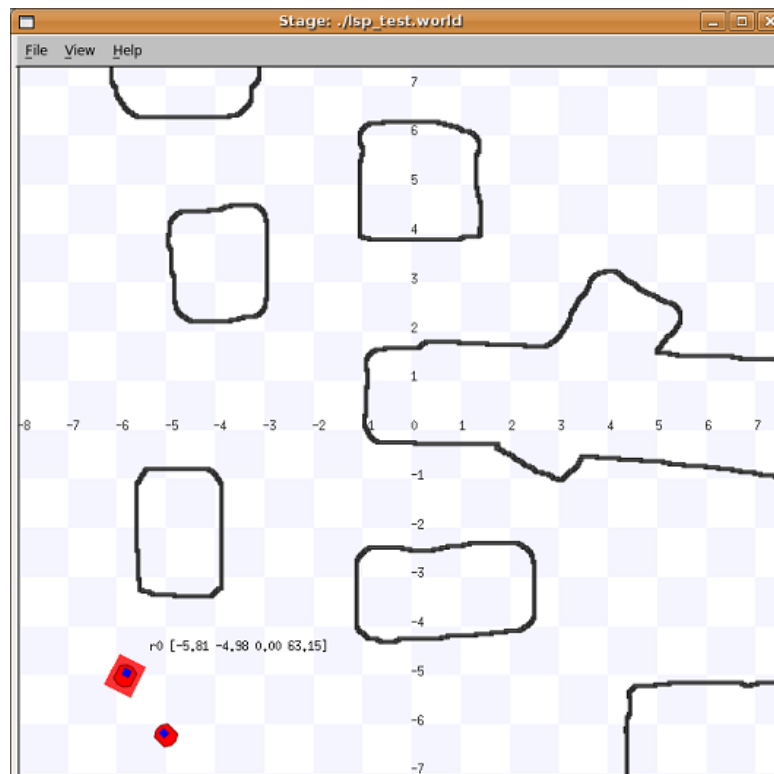


Fig. 18. Ventana del simulador Stage.

El entorno de simulación Stage funciona como un plugin de Player, es decir, como un añadido que ofrece nuevas posibilidades.

A parte de la interfaz gráfica, Stage ofrece una completa simulación de los sensores o dispositivos de un robot. Esta simulación se realiza a través de un driver llamado "stage", el cual se puede utilizar en Player como cualquier otro driver.

El driver "stage" puede proporcionar multitud de dispositivos, los cuales son accesibles a través de las interfaces estándar de Player, como en cualquier otro caso. Por tanto, la especificación del fichero de configuración de Player, es exactamente igual que si se utilizase un robot real, con la excepción de que el driver a utilizar, es el driver "stage".

4.3.2 PROCESO DE SIMULACIÓN DE STAGE

Internamente, el driver "stage" no accede a los dispositivos hardware, sino que simula dichos dispositivos. Esto es posible gracias a la arquitectura de Player, la cual permite una abstracción total a la hora de desarrollar programas, dado que no es necesario comunicarse con el hardware de forma directa, sino a través del driver. Por esto, las peticiones se lanzan contra el driver y las respuestas se reciben del driver. En un caso real, el driver retransmitiría las peticiones a los dispositivos y recibiría datos de ellos, pero en la simulación, este paso no existe. Las respuestas de los dispositivos son simuladas y no reales.

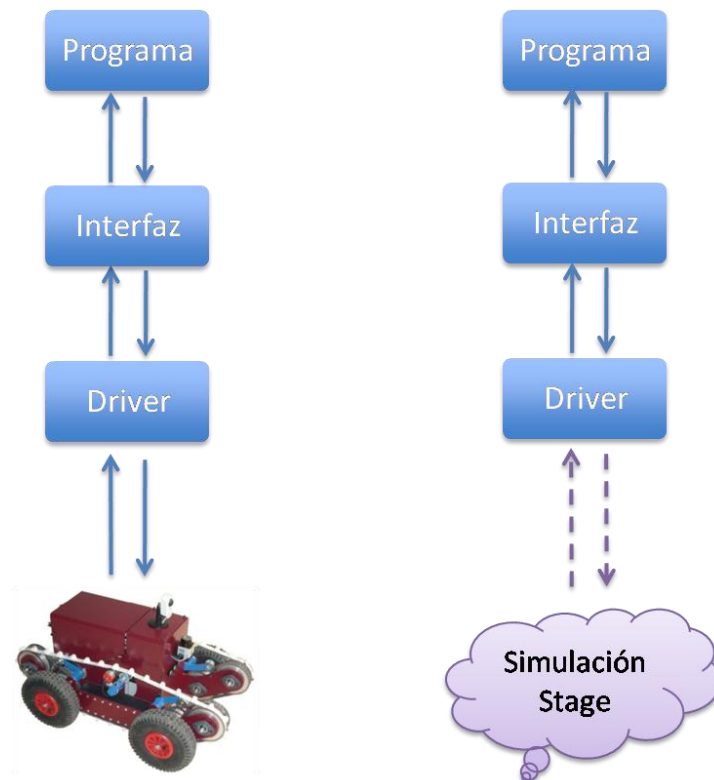


Fig. 19. Proceso real (izq.) y proceso simulado con Stage (dcha.).

El driver “stage” es el que proporciona todos los dispositivos que se quieren simular, bien sea un sensor láser o un position2d.

4.3.3 CONFIGURACIÓN DE STAGE

Para simular un robot en Player a través de Stage, es necesario definir el fichero de configuración (.cfg) de Player (con ciertas modificaciones) y un fichero .world que contiene los parámetros de la simulación.

Lo primero que se debe de hacer para realizar una simulación con Stage, es definir el fichero .world. En este fichero han de especificarse varias cosas:

- ⌘ Los modelos (o robots) que se van a simular.
- ⌘ Los dispositivos que dichos modelos van a poseer.
- ⌘ La ventana de la interfaz grafica.
- ⌘ La imagen que sirve como mapa del entorno.

Una vez se ha definido correctamente el fichero .world, sólo queda definir el fichero de configuración de Player, de forma similar a cuando se trabaja con un robot real.

A continuación se muestra un ejemplo muy básico para ilustrar la forma en la que han de escribirse los ficheros de configuración:

```
include "pioneer.inc"
include "map.inc"
include "sick.inc"

# size of the world in meters
size [16 16]

# set the resolution of the underlying raytrace model in meters
resolution 0.02

interval_sim 100
interval_real 100

# configure the GUI window
window
( size [ 695.000 693.000 ]
  center [-0.010 -0.040]
  scale 0.028 )

# load an environment bitmap
map
( bitmap "bitmaps/cave.png"
  size [16 16]
  name "cave")

# create a robot
pioneer2dx
( name "robot1"
  color "red"
  pose [-7 -7 45]
  sick_laser()
  watchdog_timeout -1.0)
```

Fig. 20. Ejemplo de fichero .world.

```
# load the Stage plugin simulation driver
driver
(
  name "stage"
  provides ["simulation:0" ]
  plugin "libstageplugin"

  # load the named file into the simulator
  worldfile "simple.world"
)

# Export the map
driver
(
  name "stage"
  provides ["map:0" ]
  model "cave"
)

# Create a Stage driver and attach position2d and
```

```
# laser interfaces to the model "robot1"
driver
(
  name "stage"
  provides ["position2d:0" "laser:0"]
  model "robot1"
)

driver
(
  name "linuxjoystick"
  provides ["joystick:0"]
  requires ["position2d:0"]
)

driver
(
  name "vfh"
  provides ["position2d:1"]
  requires ["position2d:0" "laser:0" ]
)
```

Fig. 21. Ejemplo de fichero .cfg para Stage.

Para más información sobre cómo construir los ficheros de configuración para Stage, visitar la Web del proyecto Player/Stage [2].

DESARROLLO

CAPITULO

El desarrollo es el apartado principal del proyecto, que aparte de constar de un estudio sobre las necesidades iniciales del robot Guardián y de las oportunidades que ofrece la plataforma Player/Stage para el control de robots y sus dispositivos, centra gran parte de su contenido en el diseño e implementación de la solución software que satisface las necesidades planteadas en los objetivos iniciales de este proyecto.

Como bien se ha descrito anteriormente, el principal objetivo es construir una capa de software basada en la plataforma Player/Stage, que permita el acceso a los datos de los diferentes sensores de forma sencilla y transparente, de tal forma que el hecho de que los datos provengan de una simulación o de dispositivos reales, sea indiferente. Además, esta capa debe de controlar y minimizar los posibles problemas de asincronía entre diferentes tipos de sensores, los cuales harían más complicados los posteriores tratamientos de los datos.

En los siguientes apartados, se describe todo el proceso de desarrollo de este proyecto, teniendo en cuenta los problemas que se pretenden solventar y detallando los aspectos más relevantes.

5.1 PUNTO DE PARTIDA

El estudio previo al desarrollo de los diferentes aspectos relacionados con el proyecto, como los vehículos no tripulados, los sensores, los sistemas de fusión de datos y la plataforma Player/Stage, permite localizar los problemas y las necesidades que puede tener la capa de software que se pretende implementar.

5.1.2 NECESIDADES Y OBJETIVOS

Los principales objetivos que el software debe alcanzar, los cuales han sido detallados en puntos anteriores, establecen que ha de implementarse una capa de software intermedia, entre el bajo nivel (donde se realiza la comunicación con los dispositivos) y los niveles superiores (procesamiento de los datos). Esta capa intermedia ha de implementarse sobre la plataforma Player/Stage. La función de esta capa es la de interactuar con los distintos dispositivos sensores del robot Guardián, ofreciendo métodos de acceso a los datos recogidos por los sensores.

Las principales necesidades que debe cubrir el software, se refieren a la sencillez y transparencia en el acceso a los datos recogidos por los sensores y el aislamiento de la comunicación con los dispositivos para permitir utilizar sensores reales y simulados indistintamente y sin alterar el modo de acceso a los datos.

Un aspecto importante que ha de tenerse en cuenta en el diseño de la capa de software, es la posible aparición de problemas derivados de la asincronía de los diferentes tipos de sensores [16].

5.1.1 EL PROBLEMA DE LA ASINCRONÍA

En este proyecto, la asincronía se refiere a la distinta tasa de actualización que posee cada uno de los dispositivos sensores del robot Guardián. Cada uno de los dispositivos, captura y recoge datos cada cierto tiempo, de forma periódica y constante. El tiempo que pasa entre cada captura de datos es la tasa de actualización o frecuencia y cada tipo de dispositivo tiene una específica, que no tiene por qué ser igual a las de los otros dispositivos.

El problema de la asincronía aparece cuando se intentan recuperar los datos recogidos por varios sensores a la vez y alguno de los sensores posee una tasa de actualización distinta a la de los otros dispositivos sensores. El problema radica en que los datos de aquellos sensores cuya frecuencia de actualización es menor, pueden ser datos obsoletos que no necesitan ser procesados, o por el contrario, los datos generados por aquellos sensores cuya tasa de actualización sea muy elevada, pueden perderse nunca llegar a ser procesados.

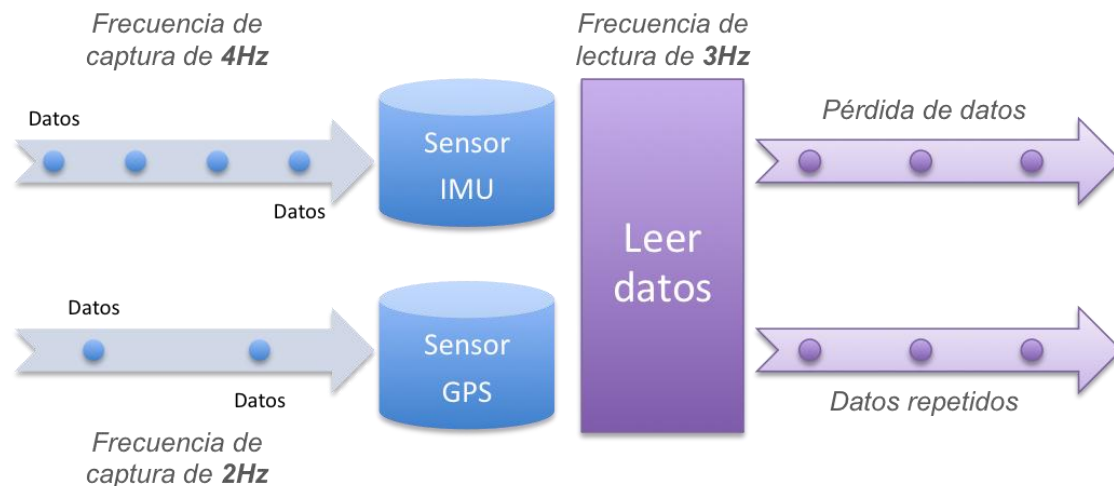


Fig. 22. Representación de la asincronía entre dos tipos de sensores.

El uso de la plataforma Player/Stage para el acceso a los dispositivos del robot, tiende a agravar el problema de la asincronía si se utilizan dispositivos sensores realmente veloces. Esto es debido a la forma en la que se accede a los dispositivos, a través de una lectura que se realiza con una frecuencia por defecto de 10Hz, que puede ser más lenta que la actualización de alguno de los sensores. Además, en el caso de las conexiones remotas vía WiFi aparece otro problema, la pérdida de paquetes que se envían a través del protocolo de red TCP/IP, muy común en las comunicaciones inalámbricas. En el capítulo 4, donde se describe el funcionamiento de Player/Stage, se describen los distintos métodos que tiene Player para realizar las lecturas de los datos, leerlos inmediatamente según son generados, o esperar a que el programa de control (cliente) los necesite. Dependiendo de la utilización de un método u otro se podrá minimizar los posibles problemas de asincronía.

Otra posible solución a los posibles problemas derivados de la asincronía de los sensores [16], a parte de las distintas opciones de lectura que ofrece Player, es la utilización de técnicas de multiprogramación. Gracias al uso de estas técnicas, se puede realizar la lectura de los datos de cada uno de los sensores en un hilo de ejecución separado, de forma que la frecuencia de lectura se aproxime a la frecuencia de actualización de cada dispositivo sensor en cada uno de los distintos casos.

Esta solución multitarea, ha de estudiarse para comprobar si realmente aporta mejoras, porque en un entorno real, como es el caso del robot Guardián, cabe la posibilidad de que todos los sensores utilizados sean más lentos que las lecturas realizada a través de Player, o que la capacidad de procesamiento de los algoritmos de más alto nivel, sea insuficiente para procesar todos los datos recogidos a la máxima frecuencia de lectura posible de Player.

Independientemente de la solución tomada con respecto a los problemas derivados de la asincronía, cabe destacar que este proyecto está dirigido hacia los sistemas de fusión de datos, para facilitar la incorporación de algoritmos ya existentes, por lo que la fiabilidad de los datos es muy importante. Estos sistemas requieren de

mucha capacidad de procesamiento, por lo que volver a procesar los mismos datos reduce la eficiencia de estos sistemas, de igual modo que no procesar algunos datos puede ser un aspecto crítico. Por esto, minimizar los problemas derivados de la asincronía de los sensores, es un aspecto muy deseable.

5.2 DISEÑO DE LA CAPA DE SOFTWARE

La capa intermedia debe de proporcionar métodos de acceso a los datos recogidos por los sensores, a la vez que aísla la forma real de obtener dichos datos, la cual puede llegar a ser complicada. Las tareas de bajo nivel, como lo son las comunicaciones entre los dispositivos sensores y el programa de control, no han de ser un impedimento a la hora de realizar el procesamiento de los datos en los sistemas de fusión de datos (u otros sistemas). Para conseguir aislar estas tareas de bajo nivel y a la vez proporcionar diversos métodos de acceso a los datos de forma sencilla, la capa de software intermedia se ha diseñado para implementar una estructura de datos en forma de lista dinámica, donde se almacenarán los datos recogidos por los sensores, y para ofrecer un conjunto de funciones para acceder a los datos de diferentes maneras, según las necesidades (datos ordenados por marca de tiempo, datos de un tipo de sensor, etc.).

En la figura 22 se ilustra un esquema genérico de la capa intermedia y su ubicación entre los diferentes niveles de un sistema real.

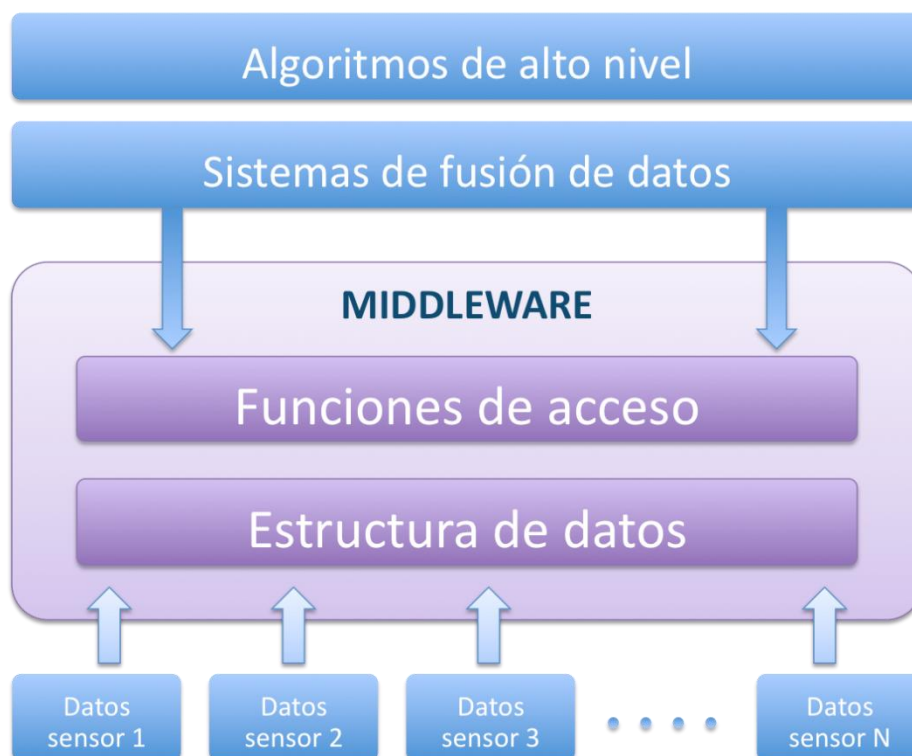


Fig. 23. Esquema de capas de un sistema de control para robots autónomos.

En la capa de más alto nivel se encuentran las rutinas o tareas que se encargan de estudiar los datos y procesar la información que estos representan. En la siguiente capa se sitúan los sistemas de preprocesamiento de datos previos al estudio de los datos en sí, generalmente son sistemas de fusión de datos de sensores que aumentan la precisión de los datos. En la capa de más bajo nivel se concentra la comunicación con el hardware para leer los dispositivos sensores y recoger sus datos.

Este proyecto tiene por objetivo la introducción de una capa intermedia que separe la comunicación con el hardware y los sistemas de preprocesamiento de datos, como los sistemas de fusión de datos de sensores, tal y como muestra la figura 22.

5.2.1 ESTRUCTURA DE DATOS DINÁMICA

La capa de software diseñada se compone principalmente por:

- ⌘ Una estructura de datos que proporciona el almacenamiento necesario para los datos recogidos por los sensores.
- ⌘ Un conjunto de funciones que proveen diferentes formas de acceso a los datos almacenados en la estructura.

La estructura de datos creada para almacenar los resultados de las lecturas de los sensores, es una estructura de tipo lista, en la cual se pueden añadir y borrar los datos. Esta estructura de tipo lista es también dinámica, en el sentido de que los datos pueden ordenarse de varias formas, pueden ser introducidos según ciertos criterios y pueden ser borrados en cualquier momento, siempre atendiendo a las necesidades de los sistemas de preprocesamiento de datos, como los sistemas de fusión de datos, y en tiempo de ejecución.

Los datos almacenados en la estructura son una encapsulación de los datos reales recogidos por los dispositivos sensores. Cada uno de estos datos contiene varios atributos específicos que permiten una gestión más flexible. Estos atributos son:

- ⌘ Información del sensor.
 - Nombre.
 - Tipo.
- ⌘ Datos recogidos por el sensor.
 - Datos en bruto.
 - Datos procesados (si procede).
- ⌘ Marca de tiempo del momento en el que los datos han sido recogidos.

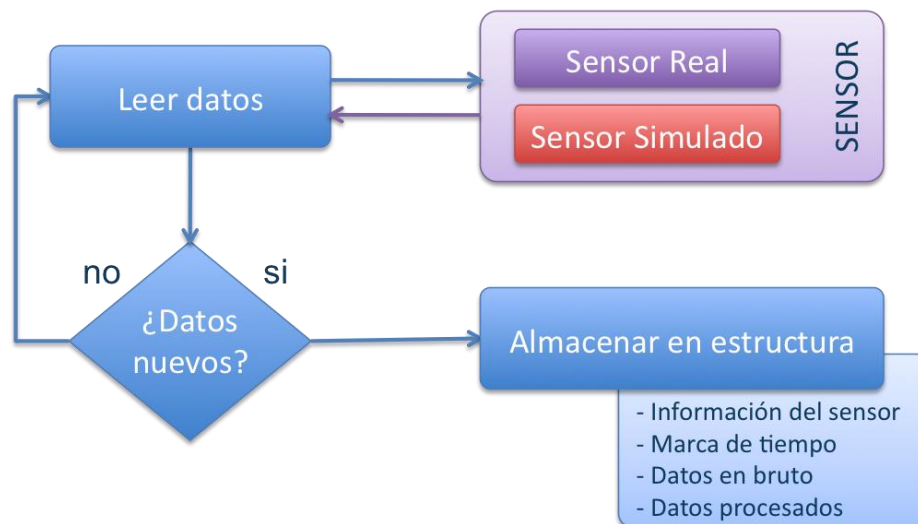


Fig. 24. Diagrama general de lectura de datos de la capa intermedia.

Respecto a los datos recogidos por algunos de los dispositivos sensores, es posible diferenciar dos clases, los datos en bruto y los datos procesados. Los datos en bruto son los valores tal y como son leídos por el dispositivo sensor, sin realizar ningún procesamiento previo, por ejemplo los valores de la aceleración en cada uno de los ejes, o la latitud medida por el GPS. Los datos procesados en cambio, corresponden a valores calculados por el propio hardware/software del sensor, a partir de los valores que obtiene en sus lecturas, como por ejemplo el valor de la posición obtenido por el sensor inercial a partir de sus distintos componentes. Como todo cálculo o procesamiento, los datos procesados contienen un cierto error (los datos en bruto también, aunque generalmente menor), pero tienen la ventaja de no necesitar ser procesados (según las necesidades). Cabe destacar que no todos los sensores son capaces de adquirir los datos en sus dos posibles formatos, pero es importante conservar ambas posibilidades cuando sea factible, ya que dependiendo de la solución implementada, serán necesarios unos u otros.

Los principales tipos de sensores que se manejan en este proyecto y son susceptibles de ser utilizados en conjunción con la capa de software intermedia, corresponden a sensores de posicionamiento GPS, sensores de odometría y sensores inerciales o IMU. Se pueden utilizar otros dispositivos, pero en este caso, en el control del robot Guardián son utilizados únicamente estos tres tipos de sensores.

5.2.2 MÉTODOS DE ACCESO A LOS DATOS

El segundo componente principal de la capa de software intermedia, es el conjunto de funciones que permiten acceder a los datos almacenados en la estructura, pudiendo acceder de distintas formas, lo que supone una mayor flexibilidad de uso.

Las funciones son diseñadas acorde a las necesidades que pueden tener los sistemas de fusión de datos, teniendo en cuenta las diferentes arquitecturas que en éstos se manejan. Por este motivo, todas y cada una de las funciones pueden devolver los datos en sus dos formas posibles, en bruto o procesados, siempre y cuando estos tipos de datos estén disponibles.

A continuación se presenta una lista con las funciones que la capa intermedia ofrece a las capas más altas para el acceso a los datos. La descripción detallada de estas funciones se encuentra en la sección 5.3.2 Detalles de la implementación de este documento.

- ⌘ Leer el primer dato disponible
- ⌘ Leer todos los datos
- ⌘ Leer sólo los datos de un tipo de sensor
- ⌘ ETC.

Gracias a estas funciones, los sistemas de procesamiento de los datos, como los sistemas de fusión de datos de sensores, pueden acceder a los datos de forma fácil y completamente transparente, de manera que no es necesario implementar directamente en los algoritmos de procesado, el código requerido para comunicarse con los sensores, el cual puede ser ciertamente complicado. Esta forma de acceso brinda la posibilidad de trabajar en entornos simulados sin tener que modificar los algoritmos de fusión de datos, por lo que es ideal para realizar pruebas y extrapolar los resultados a la realidad con una mayor precisión. Además, el uso de la estructura en forma de lista y de sus funciones de acceso, ayuda a reducir considerablemente la repetición y la pérdida de datos, causados por los problemas derivados de la asincronía entre los distintos sensores.

5.3 IMPLEMENTACIÓN DE LA CAPA DE SOFTWARE

En este apartado del documento, se presentan en detalle los aspectos más relevantes de la fase de implementación de la capa de software intermedia o middleware.

La fase de implementación supone la codificación del diseño realizado en un lenguaje de programación específico, para posteriormente ser utilizado en conjunción con los demás elementos de software que componen el sistema de control del robot Guardián. Para este proyecto, se ha utilizado el lenguaje C++ para la programación del middleware, puesto que es uno de los lenguajes que se pueden utilizar con Player/Stage y es prácticamente el más extendido, con lo que los recursos de información son bastante grandes.

5.3.1 DIAGRAMA DE CLASES

El primer paso de la fase de implementación es la construcción del diagrama de clases, que representa las relaciones entre los distintos componentes de la capa de software.

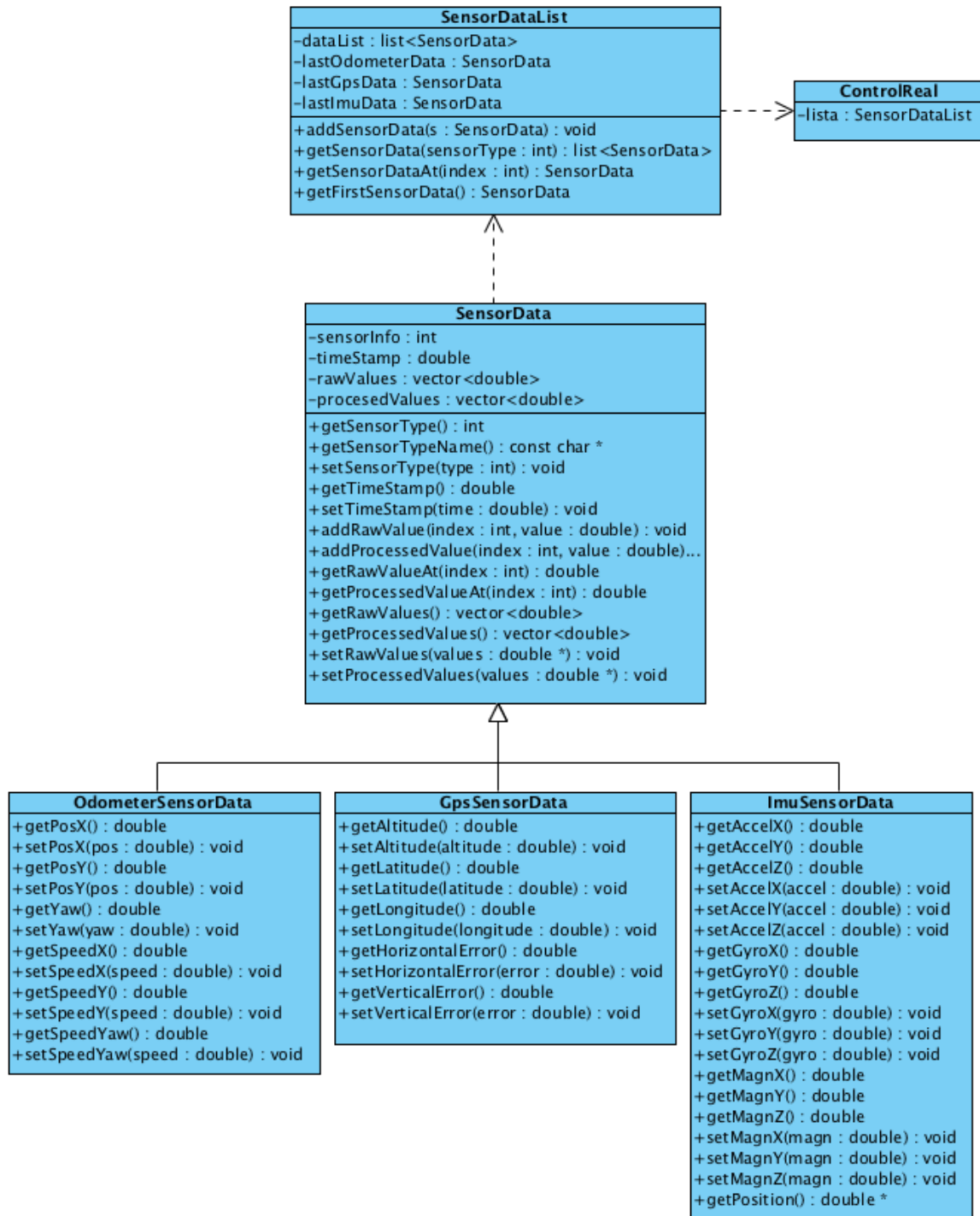


Fig. 25. Diagrama de clases del middleware.

5.3.2 DETALLES DE LA IMPLEMENTACIÓN

El diseño de clases representa los distintos elementos de los que se compone la capa de software intermedia y como se relacionan entre sí. A continuación se describe en detalle cada una de las clases, definiendo tanto sus especificaciones como su función dentro de la capa de software intermedia o middleware.

Clase SensorData

La clase SensorData representa cada uno de los datos recogidos por los sensores, añadiendo más información a parte de los propios valores de los datos. Es el elemento principal de la capa de software, el cual se va almacenar en la estructura de datos.

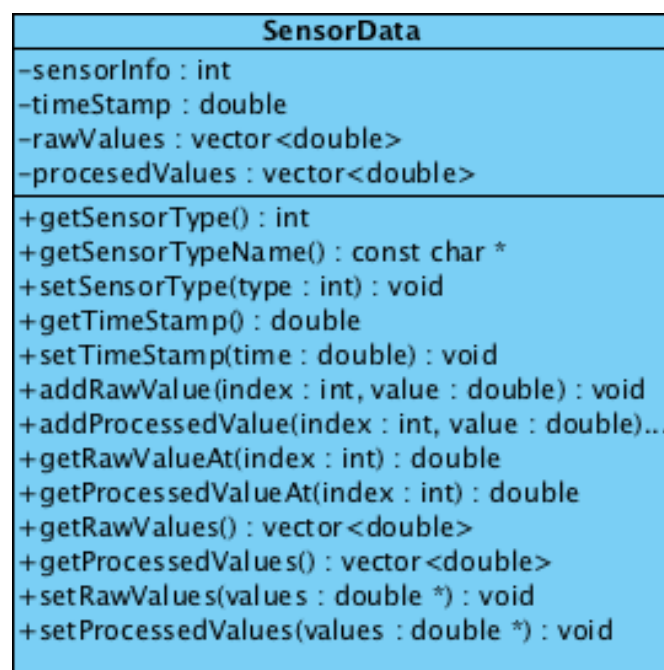


Fig. 26. Diagrama de la clase SensorData.

Esta clase es una generalización de los diferentes tipos de sensores, de la cual heredan tres clases específicas, OdometerSensorData, GpsSensorData e ImuSensorData, que poseen funciones específicas para cada tipo de datos dependiendo del sensor.

Atributos

A continuación se describen los atributos que se definen en la clase SensorData:

NOMBRE	TIPO	DESCRIPCIÓN
sensorInfo	Int	Indica el tipo de sensor que ha capturado los valores. sensorInfo puede tomar los valores: <ul style="list-style-type: none"> • ODOMETER_SENSOR • GPS_SENSOR • IMU_SENSOR
timeStamp	double	Representa la marca de tiempo del momento en el cual se ha realizado la captura de los datos por el sensor. Gracias a este atributo se pueden localizar los datos más nuevos.
rawValues	vector<double>	Es un vector que actúa como una lista de valores, el cual contiene los valores en bruto capturados por el sensor. Dependiendo del tipo de sensor, habrá más o menos valores.
processedValues	vector<double>	Es un vector que actúa como una lista de valores, el cual contiene los valores en procesados capturados por el sensor. Dependiendo del tipo de sensor, habrá más o menos valores. NOTA: No todos los sensores disponen de datos procesados, en cuyo caso este atributo no se utiliza.

Tabla 1. Atributos de la clase SensorData.

Funciones

Además de los atributos que definen a los objetos de la clase, también se definen varias funciones, las cuales se describen a continuación (no se describen los parámetros ni los tipos, únicamente el objetivo de la función):

NOMBRE	DESCRIPCIÓN
getSensorType	Devuelve el tipo de sensor del objeto SensorData, almacenado en el atributo sensorInfo.
getSensorTypeName	Devuelve el nombre del sensor, el cual es obtenido a partir del tipo. Se suele utilizar para imprimir por pantalla o fichero el nombre del sensor.
setSensorType	Modifica el tipo de sensor del objeto SensorData cambiando el atributo sensorInfo.
getTimeStamp	Devuelve la marca de tiempo en la que se capturaron los valores.
setTimeStamp	Modifica el valor de la marca de tiempo en la que se han capturado los valores, cambiando el atributo timeStamp.
addRawValue	Introduce un nuevo valor en bruto en el vector rawValues, añadiéndolo por el final.

addProcessedValue	Introduce un nuevo valor procesado en el vector processedValues, añadiéndolo por el final.
getRawValueAt	Devuelve el valor en bruto que se encuentra en una posición determinada dentro del vector de valores en bruto rawValues.
getProcessedValueAt	Devuelve el valor procesado que se encuentra en una posición determinada dentro del vector de valores procesados processedValues.
getRawValues	Devuelve el vector de valores en bruto rawValues.
getProcessedValues	Devuelve el vector de valores procesados processedValues.
setRawValues	Modifica todo el vector de valores en bruto rawValues. Resulta útil cuando se tienen todos los valores de antemano y se quieren asignar al vector rawValues.
setProcessedValues	Modifica todo el vector de valores procesados processedValues. Resulta útil cuando se tienen todos los valores de antemano y se quieren asignar al vector processedValues.

Tabla 2. Funciones de la clase SensorData.

Clase OdometerSensorData

La clase OdometerSensorData representa los datos recogidos por los sensores de tipo Odómetro. Es una especificación de la clase SensorData, con lo que hereda todos los atributos y funciones de ésta. También define funciones específicas propias de los datos recogidos por los sensores de odometría.

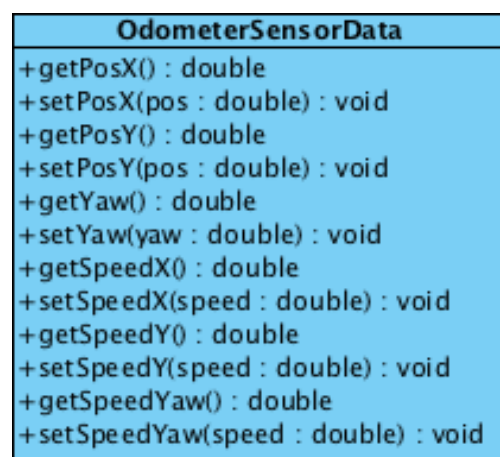


Fig. 27. Diagrama de la clase OdometerSensorData.

Atributos

La clase `OdometerSensorData` no posee atributos propios, únicamente cuenta con los definidos por la clase de la que hereda, `SensorData`, los cuales se describen en *5.3.2.1 Clase `SensorData`*.

Funciones

Además de los atributos que definen a los objetos de la clase, también se definen varias funciones, las cuales se describen a continuación (no se describen los parámetros ni los tipos, únicamente el objetivo de la función):

NOMBRE	DESCRIPCION
<code>getPosX</code>	Devuelve la coordenada X de la posición actual del robot.
<code>setPosX</code>	Modifica la coordenada X de la posición del robot. Cuando el robot cambia de posición, se puede utilizar esta función para indicarlo.
<code>getPosY</code>	Devuelve la coordenada Y de la posición actual del robot.
<code>setPosY</code>	Modifica la coordenada Y de la posición del robot. Cuando el robot cambia de posición, se puede utilizar esta función para indicarlo.
<code>getYaw</code>	Devuelve el ángulo de giro con respecto al eje vertical del robot.
<code>setYaw</code>	Modifica el ángulo de giro del robot. Cuando el robot cambia de posición, se puede utilizar esta función para indicarlo.
<code>getSpeedX</code>	Devuelve la componente X de la velocidad que lleva el robot actualmente.
<code>setSpeedX</code>	Modifica la componente X de la velocidad del robot.
<code>getSpeedY</code>	Devuelve la componente Y de la velocidad que lleva el robot actualmente.
<code>setSpeedY</code>	Modifica la componente Y de la velocidad del robot.
<code>getSpeedYaw</code>	Devuelve el ángulo de giro con respecto al eje vertical de la velocidad que lleva el robot actualmente.
<code>setSpeedYaw</code>	Modifica el ángulo de giro con respecto al eje vertical de la velocidad del robot.

Tabla 3. Funciones de la clase `OdometerSensorData`.

Clase *GpsSensorData*

La clase *GpsSensorData* representa los datos recogidos por los sensores de tipo GPS. Es una especificación de la clase *SensorData*, con lo que hereda todos los atributos y funciones de ésta. También define funciones específicas propias de los datos recogidos por los sensores de posicionamiento global o GPS.

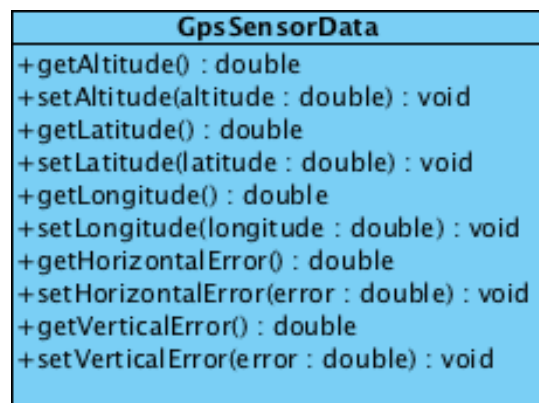


Fig. 28. Diagrama de la clase *GpsSensorData*.

Atributos

La clase *GpsSensorData* no posee atributos propios, únicamente cuenta con los definidos por la clase de la que hereda, *SensorData*, los cuales se describen en 5.3.2.1 *Clase SensorData*.

Funciones

Además de los atributos que definen a los objetos de la clase, también se definen varias funciones, las cuales se describen a continuación (no se describen los parámetros ni los tipos, únicamente el objetivo de la función):

NOMBRE	DESCRIPCION
getAltitude	Devuelve la componente de la altitud que identifica a la posición actual del robot.
setAltitude	Modifica la componente de la altitud.
getLatitude	Devuelve la componente de la latitud que identifica a la posición actual del robot.
setLatitude	Modifica la componente de la latitud.

getLongitude	Devuelve la componente de la longitud que identifica a la posición actual del robot.
setLongitude	Modifica la componente de la longitud.
getHorizontalError	Devuelve el valor del error en el eje horizontal con respecto a la posición actual del robot calculada por el sensor GPS.
setHorizontalError	Modifica el valor del error en el eje horizontal con respecto a la posición actual del robot calculada por el sensor GPS.
getVerticalError	Devuelve el valor del error en el eje vertical con respecto a la posición actual del robot calculada por el sensor GPS.
setVerticalError	Modifica el valor del error en el eje vertical con respecto a la posición actual del robot calculada por el sensor GPS.

Tabla 4. Funciones de la clase GpsSensorData.

Clase ImuSensorData

La clase ImuSensorData representa los datos recogidos por los sensores de tipo inercial o IMU. Es una especificación de la clase SensorData, con lo que hereda todos los atributos y funciones de ésta. También define funciones específicas propias de los datos recogidos por los sensores inerciales o IMU.

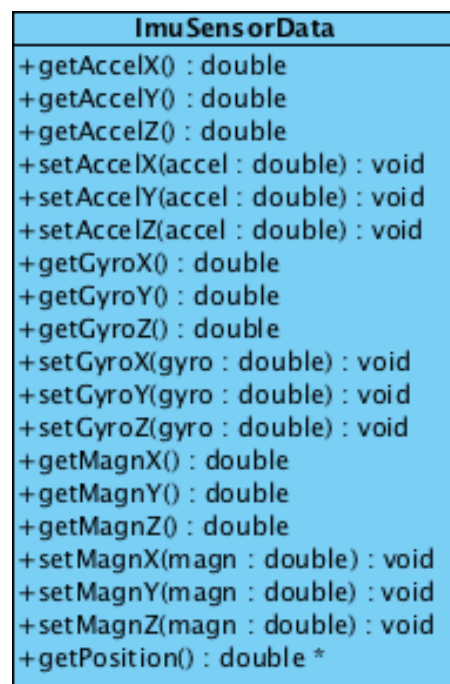


Fig. 29. Diagrama de la clase ImuSensorData.

Atributos

La clase `ImuSensorData` no posee atributos propios, únicamente cuenta con los definidos por la clase de la que hereda, `SensorData`, los cuales se describen en 5.3.2.1 *Clase `SensorData`*.

Funciones

Además de los atributos que definen a los objetos de la clase, también se definen varias funciones, las cuales se describen a continuación (no se describen los parámetros ni los tipos, únicamente el objetivo de la función):

NOMBRE	DESCRIPCION
<code>getAccelX</code>	Devuelve la componente X de la aceleración medida por el sensor IMU.
<code>setAccelX</code>	Modifica la componente X de la aceleración.
<code>getAccelY</code>	Devuelve la componente Y de la aceleración medida por el sensor IMU.
<code>setAccelY</code>	Modifica la componente Y de la aceleración.
<code>getAccelZ</code>	Devuelve la componente Z de la aceleración medida por el sensor IMU.
<code>setAccelZ</code>	Modifica la componente Z de la aceleración.
<code>getGyroX</code>	Devuelve la componente del ángulo de giro sobre el eje X medida por el sensor IMU.
<code>setGyroX</code>	Modifica la componente del ángulo de giro sobre el eje X.
<code>getGyroY</code>	Devuelve la componente del ángulo de giro sobre el eje Y medida por el sensor IMU.
<code>setGyroY</code>	Modifica la componente del ángulo de giro sobre el eje Y.
<code>getGyroZ</code>	Devuelve la componente del ángulo de giro sobre el eje Z medida por el sensor IMU.
<code>setGyroZ</code>	Modifica la componente del ángulo de giro sobre el eje Z.
<code>getMagnX</code>	Devuelve la componente X del magnetómetro medida por el sensor IMU.
<code>setMagnX</code>	Modifica la componente X del magnetómetro.
<code>getMagnY</code>	Devuelve la componente Y del magnetómetro medida por el sensor IMU.
<code>setMagnY</code>	Modifica la componente Y del magnetómetro.
<code>getMagnZ</code>	Devuelve la componente Z del magnetómetro medida por el sensor IMU.
<code>setMagnZ</code>	Modifica la componente Z del magnetómetro.

getPosition	Devuelve la posición integrada obtenida a partir de los datos de los sensores de la IMU. Concretamente se devuelve un array con los valores de yaw, pitch y roll.
-------------	---

Tabla 5. Funciones de la clase ImuSensorData.

Clase SensorDataList

La clase SensorDataList representa una estructura de datos en forma de lista, en la cual se van a almacenar los datos recogidos por los sensores. Cada elemento almacenado corresponderá a un objeto o instancia de alguna de las clases específicas que heredan de la clase SensorData, de forma que para cada dato, se almacenen sus valores en bruto, sus valores procesados (si procede), la información del sensor que los ha capturado y una marca de tiempo del momento de la captura de los valores.

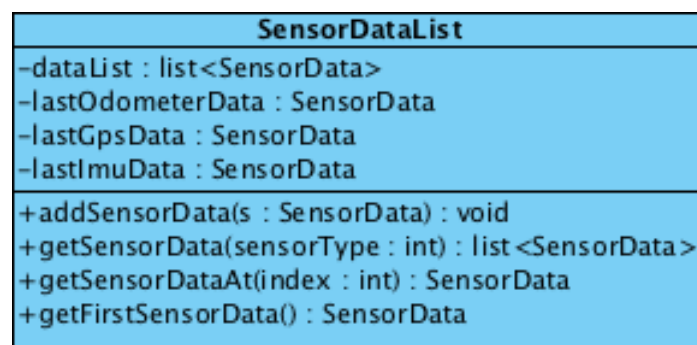


Fig. 30. Diagrama de la clase SensorDataList.

Atributos

A continuación se describen los atributos que definen a la clase SensorDataList:

NOMBRE	TIPO	DESCRIPCION
dataList	List<SensorData>	Estructura de tipo lista que almacena objetos de tipo SensorData. Se trata de la estructura dinámica de almacenamiento del middleware para almacenar los datos leídos por los sensores.
lastOdometerData	SensorData	Último valor leído por el sensor Odómetro. Se utiliza para realizar una comparación y cerciorarse de que los datos que se almacenan no son obsoletos o duplicados.

lastGpsData	SensorData	Último valor leído por el sensor GPS. Se utiliza para realizar una comparación y cerciorarse de que los datos que se almacenan no son obsoletos o duplicados.
lastImuData	SensorData	Último valor leído por el sensor IMU. Se utiliza para realizar una comparación y cerciorarse de que los datos que se almacenan no son obsoletos o duplicados.

Tabla 6. Atributos de la clase SensorDataList.

Funciones

Además de los atributos que definen a los objetos de la clase, también se definen varias funciones, las cuales se describen a continuación (no se describen los parámetros ni los tipos, únicamente el objetivo de la función):

NOMBRE	DESCRIPCION
addSensorData	Añade un nuevo objeto de tipo SensorData a la lista de datos, el cual contiene los valores leídos por un sensor, además de otra información.
getSensorData (type)	Obtiene todos los objetos de tipo SensorData de la lista de datos, que pertenecen a un tipo específico de sensor, representado por el parámetro 'type'.
getSensorDataAt	Obtiene el objeto de tipo SensorData almacenado en una posición específica dentro de la lista de datos. La posición se pasa como parámetro.

Tabla 7. Funciones de la clase SensorDataList.

EXPERIMENTACIÓN

CAPITULO

En este capítulo se describe la experimentación llevada a cabo en el proyecto. La experimentación se refiere a las pruebas realizadas para medir la capacidad y la eficiencia de la capa de software desarrollada, así como para comprobar el funcionamiento general del robot Guardián y de sus sensores. Gracias a las pruebas realizadas sobre distintos entornos y con diferentes configuraciones, se pueden sacar ciertas conclusiones y realizar valoraciones que posteriormente pueden ser fundamentales para el éxito del proyecto y de los futuros trabajos apoyados en él.

La experimentación llevada a cabo en este proyecto consiste en una batería de pruebas para comprobar el comportamiento del robot y de sus sensores, sobre todo del sensor inercial, sobre un entorno simulado y posteriormente sobre el robot Guardián. En cada una de las pruebas, el robot realiza un movimiento controlado durante un tiempo predeterminado, siendo este movimiento diferente para cada prueba (trayectorias curvas, rectas, etc.).

Para llevar a cabo las pruebas con un mínimo de rigor y fiabilidad, se ha optado por programar los movimientos directamente en el programa de control, el cual implementa la capa de software intermedia para la gestión de los datos de los sensores. Por motivos de compatibilidad, únicamente es posible simular el sensor Odómetro en Stage, por lo que la IMU y el GPS no pueden ser comparados con los resultados simulados. Las pruebas con estos sensores se dirigen a observar su funcionamiento y fiabilidad. Cabe destacar que el GPS no ha funcionado correctamente durante la realización de este proyecto, por lo que no se ha podido probar su funcionamiento.

6.1 INTRODUCCIÓN Y PREPARACIÓN DE LAS PRUEBAS

Tal y como se ha planteado en puntos anteriores de este documento, existe un problema denominado asincronía entre sensores, el cual puede ocasionar repetición de datos o pérdida de los mismos. Para resolver este problema se sugiere utilizar técnicas de multiprogramación para realizar la lectura de los datos de los sensores de una manera más eficiente y fiable. Antes de valorar si es necesario implementar este tipo de técnicas, se han realizado las pruebas de forma normal, sin recurrir a la paralelización [17], ya que es muy posible que no sea necesario utilizar dichas técnicas, debido a que las necesidades actuales pueden ser cubiertas con la velocidad de lectura de Player (10 Hz) y únicamente bastaría con realizar una comparación para evitar datos repetidos.

Respecto al escenario sobre el cual se han realizado las pruebas, cabe destacar que se ha utilizado una conexión inalámbrica (Wifi) para controlar el robot, de forma que las primeras pruebas se han ejecutado desde un PC externo, con sistema operativo Ubuntu 9.10, conectado al servidor Player que se ejecuta sobre el robot. Las pruebas se han realizado en la sala del laboratorio del GIAA de la Universidad Carlos III de Madrid (Colmenarejo).

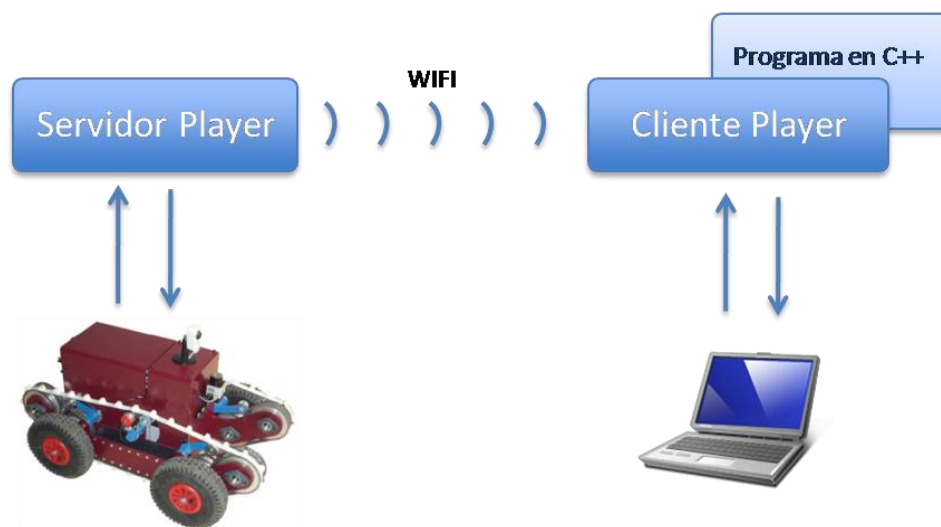


Fig. 31. Esquema de conexión entre el robot y el PC.

El robot Guardián cuenta con un punto de acceso que crea una red inalámbrica llamada "guardian". El PC interno del robot se conecta a ese punto de acceso en la dirección 192.168.2.11 por defecto. El PC que actúa de cliente se conecta a la red "guardian" creada por el punto de acceso del robot con el fin de poder conectarse posteriormente con el servidor Player que se encuentra corriendo en el robot.

Las pruebas se han realizado con el programa de control adjunto a este documento, el cual hace uso de la capa de software intermedia desarrollada e incorpora una sencilla detección de obstáculos mediante el uso del sensor láser [18]. Este programa permite al usuario mover el robot de forma manual, utilizando el teclado o el joystick. Para las pruebas realizadas, se ha desactivado el control manual y se ha codificado el recorrido que el robot ha de seguir en cada prueba, de forma que las pruebas puedan ser reproducibles y fiables.

Antes de comenzar a realizar pruebas, primero se ha estudiado qué tipo de pruebas son más representativas y cubren el mayor rango de posibles escenarios. Las pruebas se centran en realizar distintos recorridos con el robot en el entorno simulado y posteriormente en el entorno real.

Para el entorno simulado únicamente se utilizan el sensor odómetro, ya que ni el sensor inercial, ni el sensor GPS están soportado por Stage, por lo que los datos recogidos por la IMU y el GPS se utilizarán únicamente para analizar el comportamiento de estos sensores.

6.2 PRUEBAS CON TRAYECTORIAS RECTILÍNEAS

Las primera pruebas que se han llevado a cabo, consisten en que el robot realice diferentes trayectorias rectilíneas, como moverse en línea recta, realizar un movimiento en forma de “L”, etc. En estas pruebas puede comprobarse el comportamiento del sensor odómetro, así como el funcionamiento del sensor inercial ante giros bruscos de 90 grados.

Cada una de las pruebas consta de los resultados de la simulación del sensor Odómetro en Stage y de los resultados del sensor Odómetro real y de la IMU, ambos dispositivos del robot Guardián.

PRUEBA 1

En esta prueba se ha realizado un recorrido en línea recta, desde el principio hasta el final. Con esta prueba se pretende observar si el odómetro es capaz de representar el movimiento recto fielmente, o si por el contrario comete un error en la medición y en ése caso poder cuantificarlo. A continuación se representan gráficamente los datos para su comparación y observación.



Fig. 32. Recorrido rectilíneo a realizar en la prueba.

A continuación se muestran los resultados obtenidos en esta prueba, de forma gráfica. La gráfica representa el plano del suelo en sus dos dimensiones.

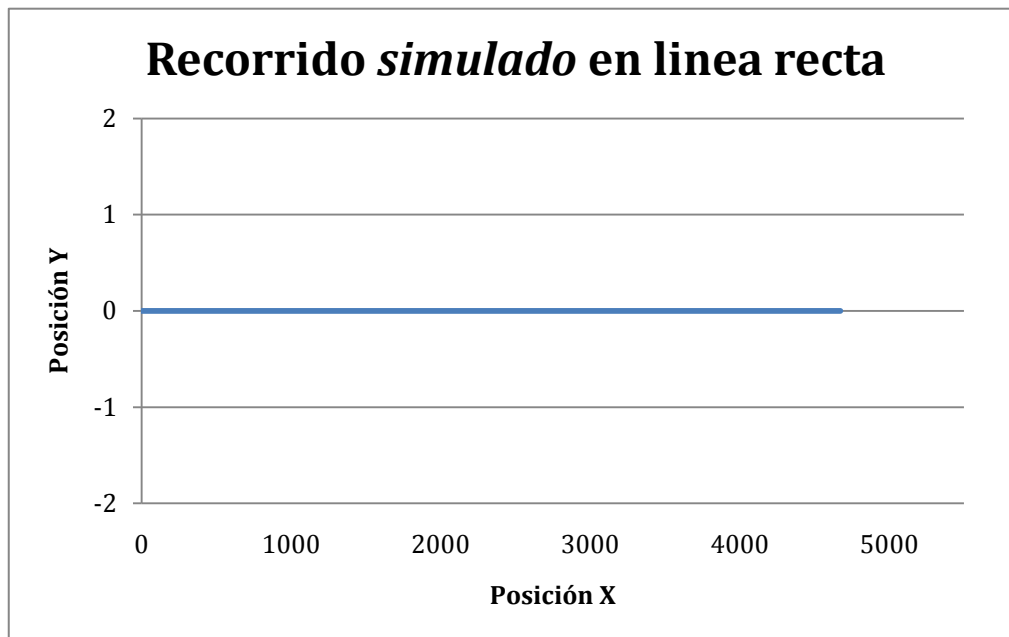


Fig. 33. Representación de los datos simulados.

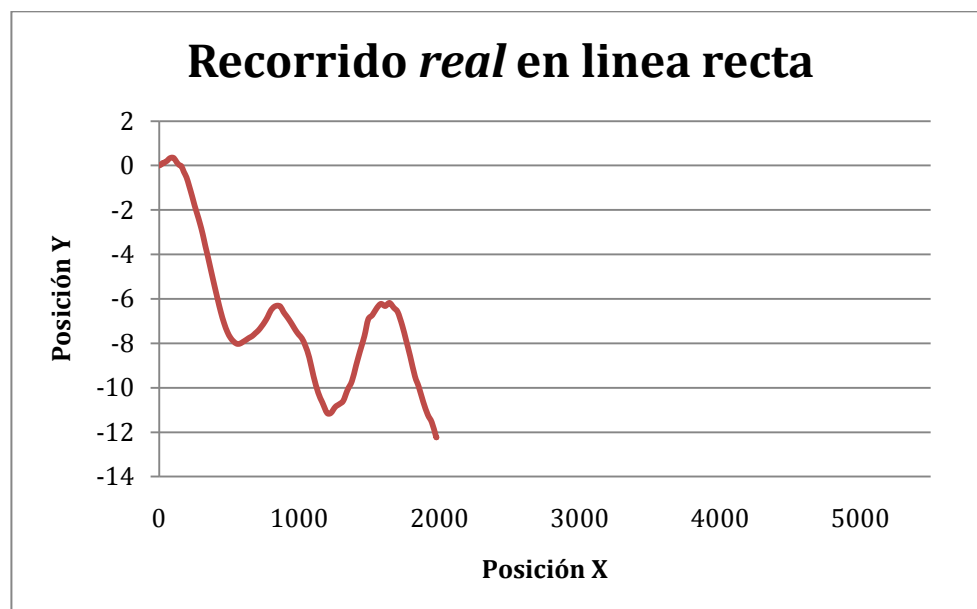


Fig. 34. Representación de los datos reales.

Los datos obtenidos a partir de la simulación en Stage son bastante buenos, ya que es posible observar que el robot ha seguido una línea recta perfectamente. En el caso de los datos reales, el simple hecho de realizar un recorrido en línea recta, basta

para que el odómetro del robot no sea capaz de medir con la misma exactitud de la simulación, como era de esperar. Como puede observarse en la gráfica, el error del odómetro se sitúa en torno a 14 puntos. Si bien parece que los resultados son muy malos, hay que destacar que la escala del eje Y es mucho menor que la del eje X, con lo que si se igualan las escalas (fig. 24), el recorrido realizado por el robot real, se asemeja a una línea recta.

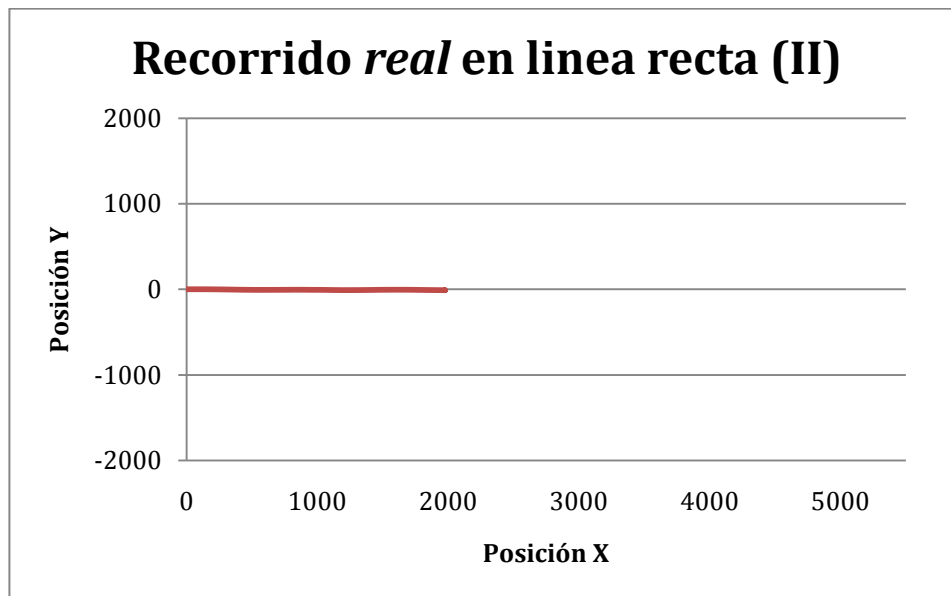


Fig. 35. Representación de los datos reales normalizando la escala de los ejes.

Durante la simulación se han recogido únicamente los datos del sensor de tipo Odómetro, ya que los demás sensores no pueden ser simulados. De todas formas, en la realización de la prueba sobre el entorno real, sí se han recogido los datos proporcionados por la IMU, además de los del sensor Odómetro del robot Guardián.

A continuación se describen los resultados obtenidos sobre el entorno real, en lo que a los datos recogidos por el sensor inercial se refiere.

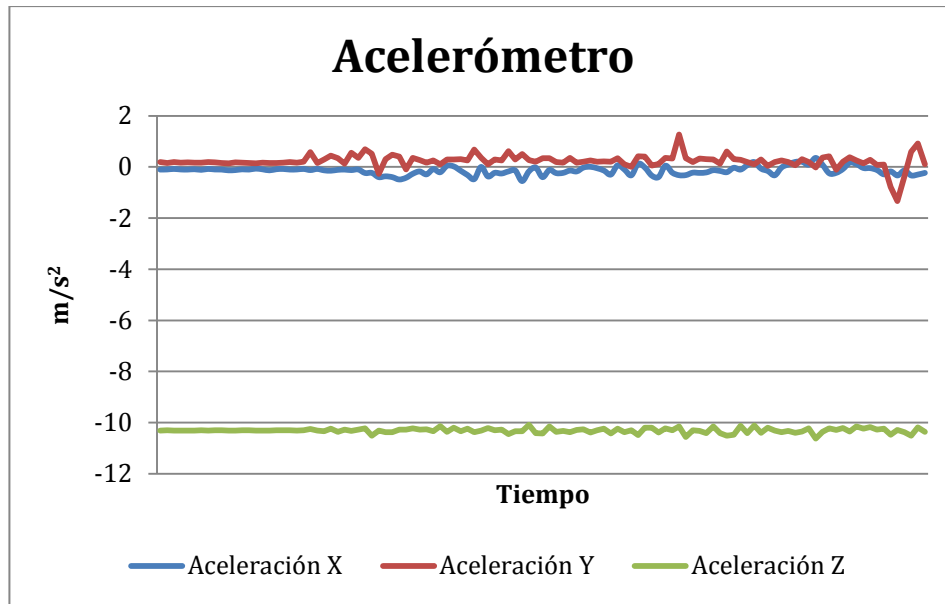


Fig. 36. Datos obtenidos de la IMU (Acelerómetro).

Las primeras pruebas realizadas con el sensor inercial, indican que la IMU tiene un ruido muy elevado, lo que la hace prácticamente inservible si no se calibra o se preprocesa de alguna manera. Las gráficas de arriba representan en su primer cuarto al robot parado, justo antes de empezar a realizar el movimiento en línea recta.

Como se puede observar, aún estando el robot parado, el acelerómetro indica aceleraciones en el eje X e Y cercanas a $0.3 \text{ (m/s}^2\text{)}$ y para el eje Z, la fuerza de la gravedad, el sensor indica valores en torno a $-10.7 \text{ (m/s}^2\text{)}$, los cuales se alejan mucho de la realidad, teniendo en cuenta que el robot se encuentra parado. Del primer cuarto en adelante el robot se mueve en línea recta, por eso se pueden apreciar ligeras variaciones en los sensores de la IMU.

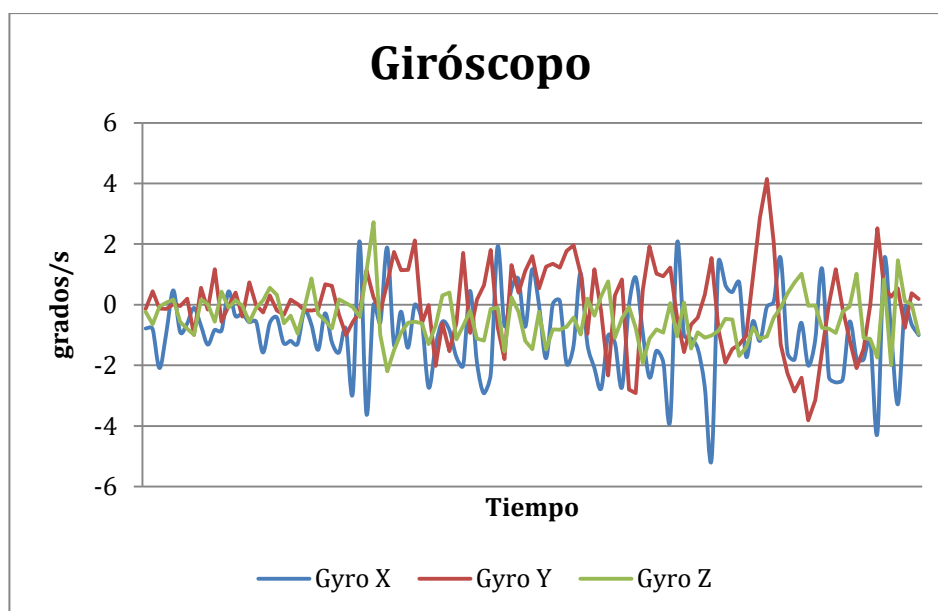


Fig. 37. Datos obtenidos de la IMU (Gyro).

El acelerómetro de la IMU muestra un ruido muy elevado que impide apreciar con claridad y exactitud los distintos movimientos realizados por el robot Guardián. En lo que respecta al giróscopo, pasa exactamente lo mismo, existe un ruido en las mediciones excesivamente elevado, cercano a los 2 grados/s en ciertas ocasiones. Basta con observar la gráfica (fig. 31) para darse cuenta de la poca fiabilidad del sensor inercial.

A tenor de los resultados arrojados por el acelerómetro y el giróscopo, es muy posible que el sensor inercial esté descalibrado o que necesite un postprocesamiento para filtrar el ruido.

PRUEBA 2

En esta prueba se ha realizado un recorrido en forma de "L", desde el principio hasta el final, realizando un giro de 90 grados entre medias. Con esta prueba se pretende observar si el odómetro es capaz de representar el movimiento recto y el giro brusco fielmente, o si por el contrario comete un error en la medición y en ese caso poder cuantificarlo. A continuación se representan gráficamente los datos para su comparación y observación.



Fig. 38. Recorrido rectilíneo a realizar en la prueba.

A continuación se muestran los resultados obtenidos en esta prueba, de forma gráfica. La gráfica representa el plano del suelo en sus dos dimensiones.

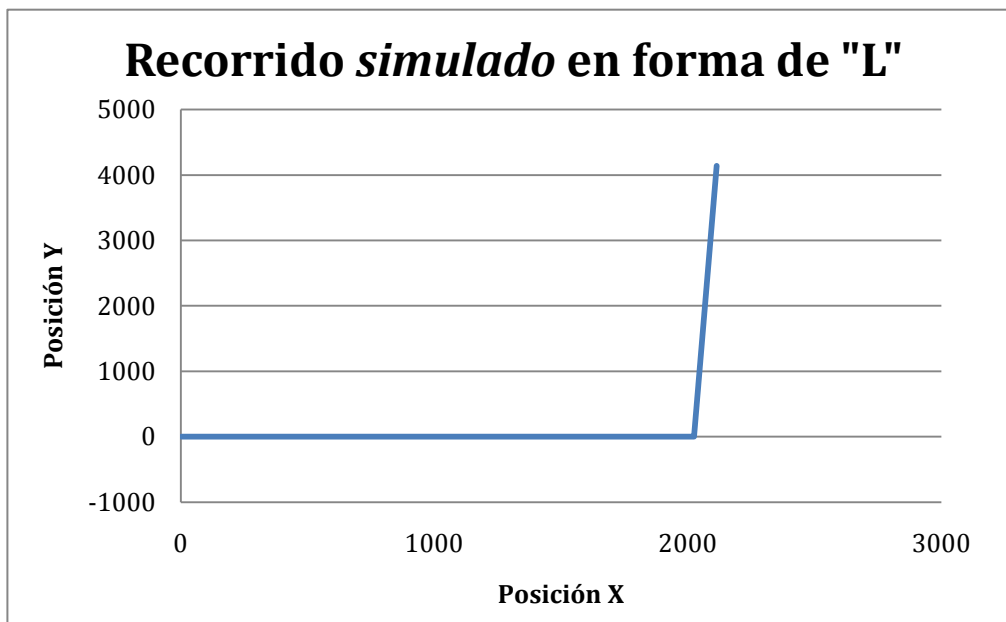


Fig. 39. Representación de los datos simulados.

En esta segunda prueba, en la cual se ha realizado un recorrido en forma de “L”, los datos arrojados por la simulación son casi perfectos, al igual que sucedió en la prueba 1. Por el contrario, los datos obtenidos del robot real, muestran un escaso acierto a la hora de realizar el recorrido (fig. 27). Es muy probable que la poca precisión del sensor Odómetro del robot Guardián, se deba al hecho de que se realiza un giro brusco de 90 grados sobre el eje vertical del robot. Puesto que el robot Guardián posee un movimiento de tipo diferencial, el rápido giro de 90 grados no es correctamente medido por el Odómetro, suponiendo que el sensor está estropeado.

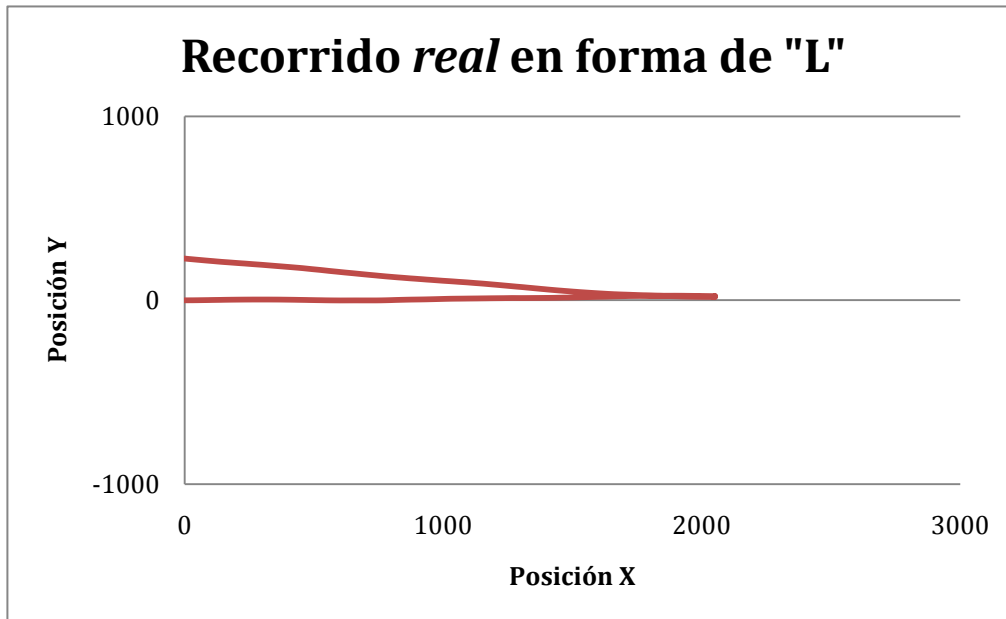


Fig. 40. Representación de los datos reales.

Una vez observados los datos arrojados por la prueba 2, en lo que al sensor de tipo Odómetro se refiere, se procede a examinar los resultados obtenidos mediante el sensor inercial del que dispone el robot Guardián. En la primera prueba se ha descubierto que este sensor introduce un ruido muy grande en los datos medidos, tanto en los componentes del acelerómetro, como en los del giróscopo. En esta segunda prueba se pretende observar el comportamiento de la IMU ante cambios de dirección repentinos y comprobar si dichos cambios son registrados y verificables a pesar del enorme ruido en los datos.

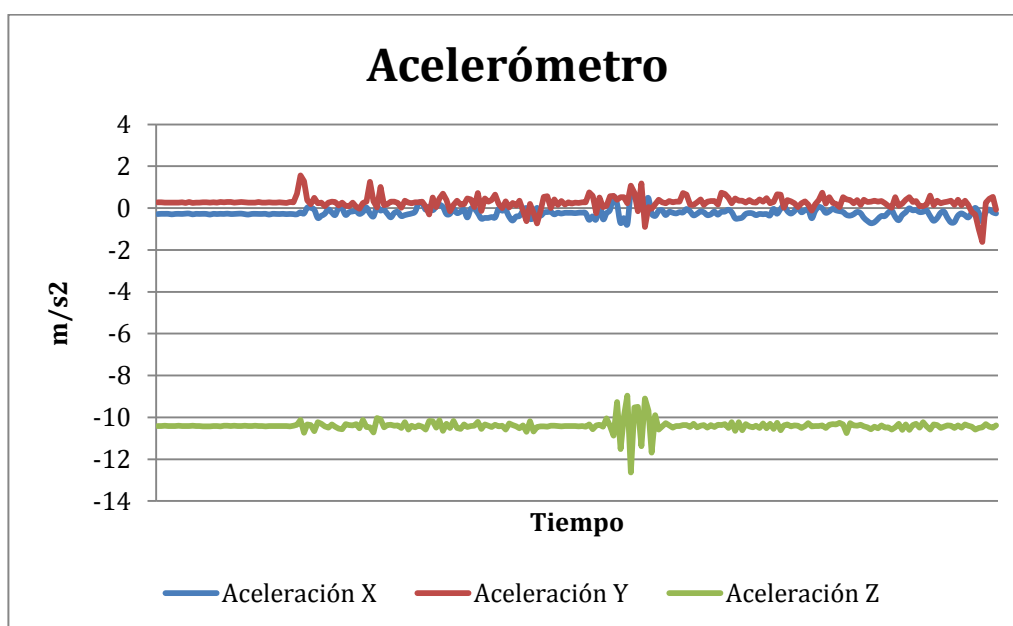


Fig. 41. Datos de la IMU (Acelerómetro).

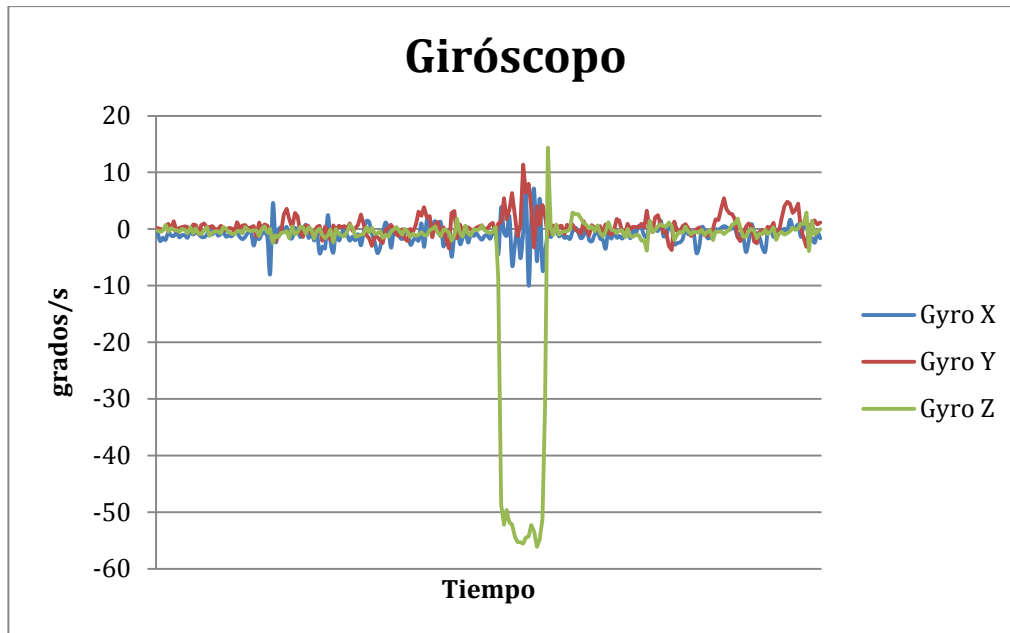


Fig. 42. Datos de la IMU (Gyro).

En esta segunda prueba, en la que se ha realizado un recorrido en forma de “L”, se puede observar como el sensor giróscopo capta la rotación en el eje vertical (Z) en el momento en el que el robot gira un ángulo de 90 grados para completar el recorrido en forma de “L”. El giróscopo registra un movimiento angular de 60 grados/s aproximadamente durante un pequeño periodo de tiempo, hasta completar el giro de 90 grados. Lo que no tiene mucha explicación es la fluctuación de la aceleración en el eje Z casi en el mismo momento de la rotación, aunque quizás pueda deberse a las vibraciones propias del robot cuando éste está en movimiento.

A tenor de los resultados que se han obtenido en estas dos pruebas, es posible que el dispositivo odómetro del robot real, o su software, no funcione correctamente, posiblemente debido al giro de tipo diferencial que posee el robot, aunque es necesario realizar más pruebas para determinar este hecho. En cuanto al sensor inercial, es imprescindible reducir el ruido que se observa en las mediciones, ya que impide totalmente la interpretación de manera fiable de los datos. De todas formas, parece ser que el giróscopo funciona mejor que el acelerómetro, ya que registra con claridad el movimiento giratorio del robot Guardián.

6.3 PRUEBAS CON TRAYECTORIA CURVAS

En esta y en sucesivas pruebas, se ha optado por incluir una trayectoria curvilínea preprogramada, de forma que la ejecución será siempre igual, ya que no interviene la acción del usuario en ningún momento. Durante toda la trayectoria el robot avanza y gira a la vez de manera constante, durante un determinado tiempo.

En cada una de las diferentes pruebas realizadas, el robot realiza un movimiento en forma de curva suave, primero con un ángulo de giro pequeño, el cual se va aumentando en cada una de las pruebas, hasta lograr que el robot realice más de un círculo completo.

Cada una de las pruebas consta de los resultados de la simulación del sensor Odómetro en Stage y de los resultados del sensor Odómetro real y de la IMU, ambos dispositivos del robot Guardián.

PRUEBA 1

Para la primera trayectoria, se ha utilizado un ángulo de giro bastante pequeño para intentar observar si los cambios más pequeños en la orientación del robot, son detectados por la IMU o por el contrario pasan inadvertidos.

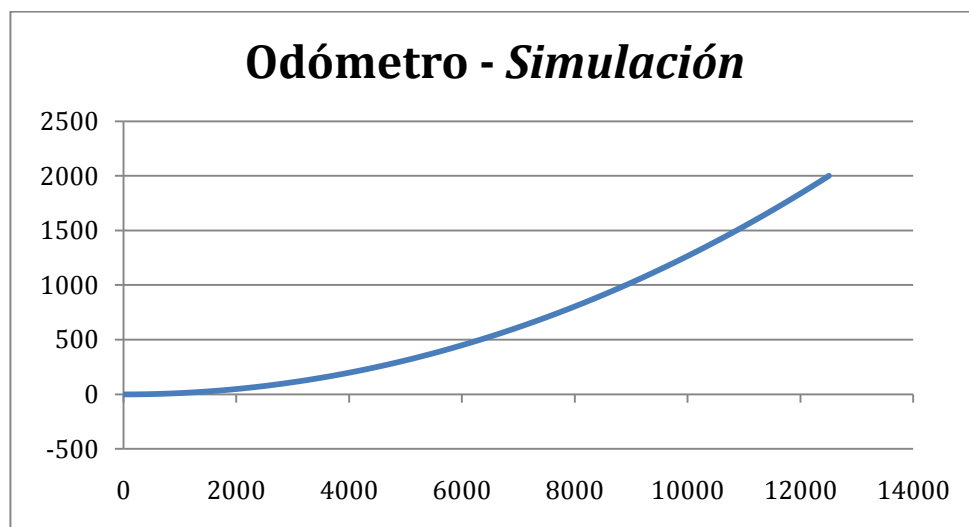


Fig. 43. Trayectoria simulada

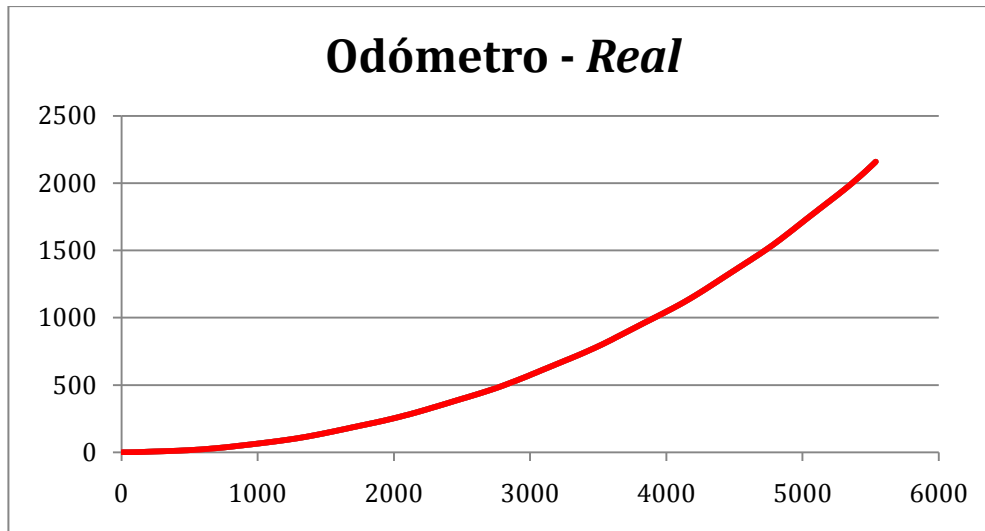


Fig. 44. Trayectoria real

A tenor de los resultados obtenidos en esta primera prueba de trayectorias circulares, podemos ver que el odómetro se comporta bastante bien, reflejando fielmente la trayectoria realizada, la cual es muy semejante a la realizada en la simulación. Estos datos son claramente mejores que los obtenidos en las pruebas de control manual, si bien es cierto que en esta prueba el robot no realiza giros bruscos.

A continuación se presentan los datos obtenidos por la IMU.

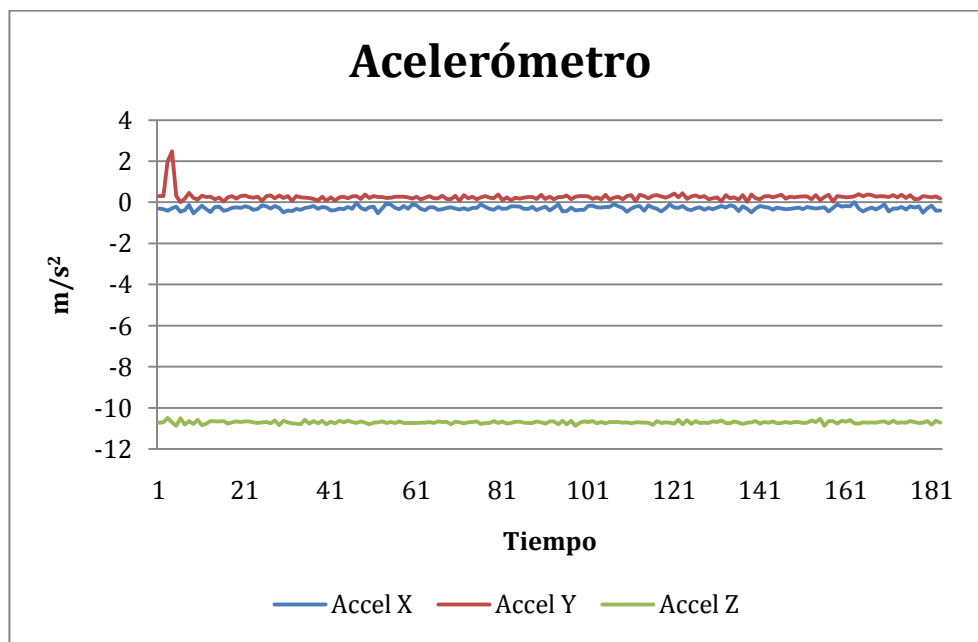


Fig. 45. Datos de la IMU (Aceleración).

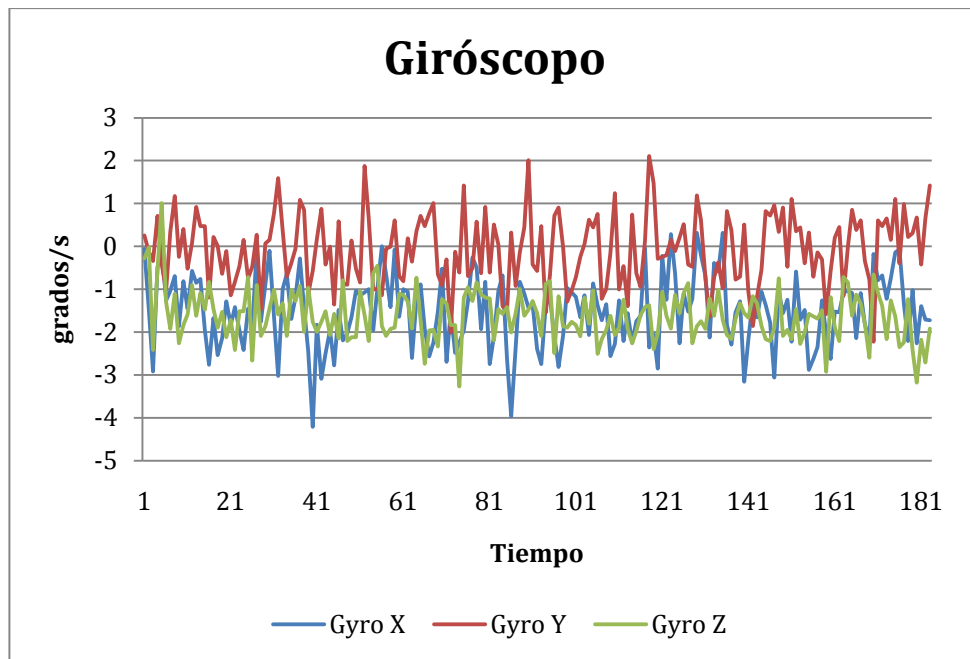


Fig. 46. Datos de la IMU (Gyro).

Observando los datos capturados por el sensor inercial, se aprecia que el ruido es bastante grande y continuo, aunque no constante, lo que impide realizar una corrección de la desviación de manera sencilla.

El acelerómetro si capta el momento en el que el robot empieza a moverse, pero el giróscopo no muestra indicios de que el robot esté realizando un giro. Esto puede ser culpa del ruido que impide apreciar el pequeño ángulo de giro, el cual debería mostrarse constante.

PRUEBA 2

En la siguiente prueba de trayectoria curva, se ha utilizado un ángulo de giro un poco más grande que en la primera prueba. Analizando los resultados obtenidos a partir de los datos capturados por los sensores, podremos sacar ciertas conclusiones.

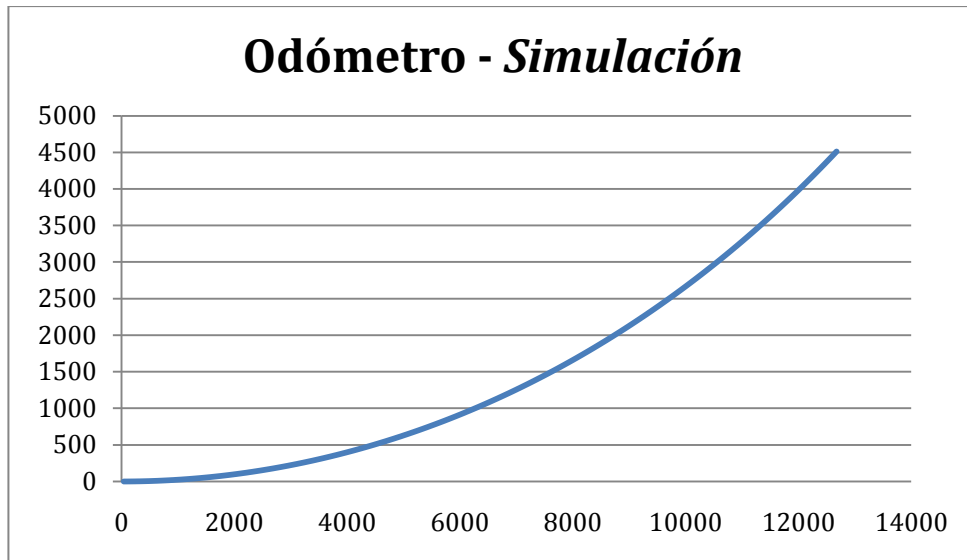


Fig. 47. Trayectoria simulada

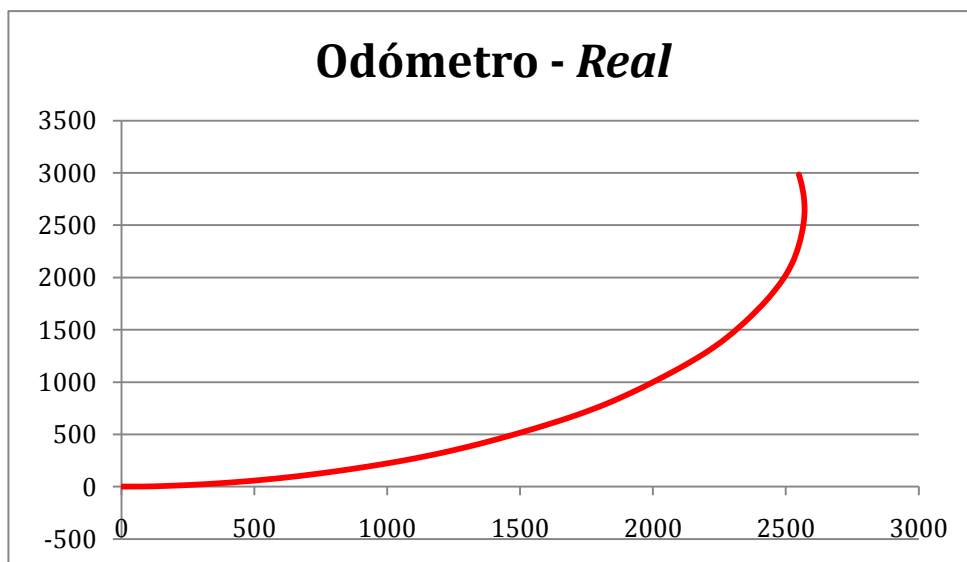


Fig. 48. Trayectoria real

En esta ocasión la trayectoria capturada por el odómetro del robot real, no se ajusta tan bien a la trayectoria simulada como en la prueba anterior. Este hecho genera cierta confusión, puesto que el ángulo de giro es constante durante todo el recorrido, así como la velocidad, por lo que no debería producirse un giro tan pronunciado hacia el final del recorrido. En este caso los datos capturados por el odómetro no son muy fiables.

Respecto a los datos capturados por el sensor inercial, se puede comprobar que en esta ocasión el giróscopo ha recogido datos que indican que se ha producido un movimiento angular sobre el eje Z. A pesar del ruido de los datos, es posible apreciar que en el eje Z se produce un giro continuo de 4 o 5 grados aproximadamente. Por

otro lado, el acelerómetro ha captado la aceleración inicial correspondiente al movimiento del robot en los primeros segundos.

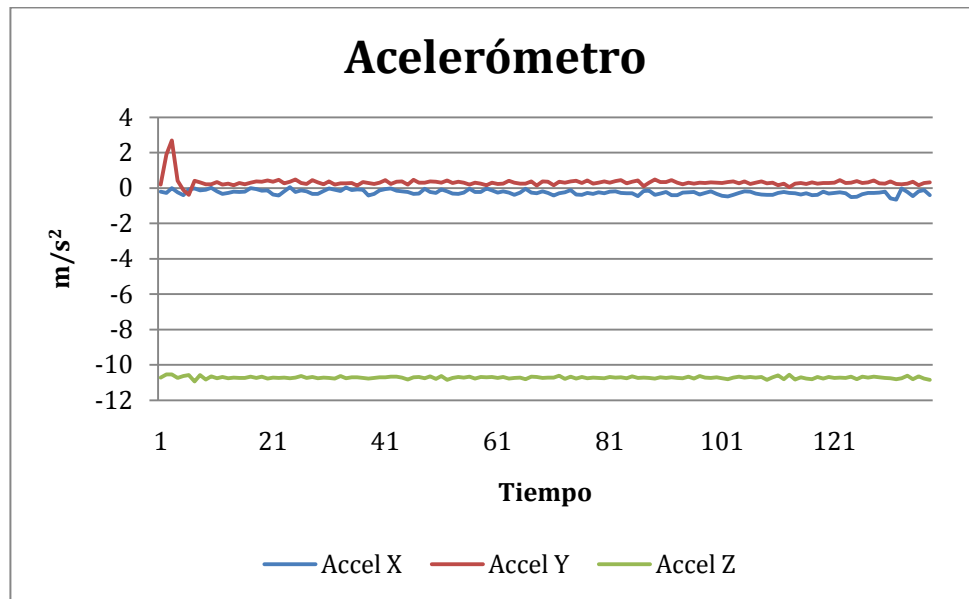


Fig. 49. Datos de la IMU (Aceleración).

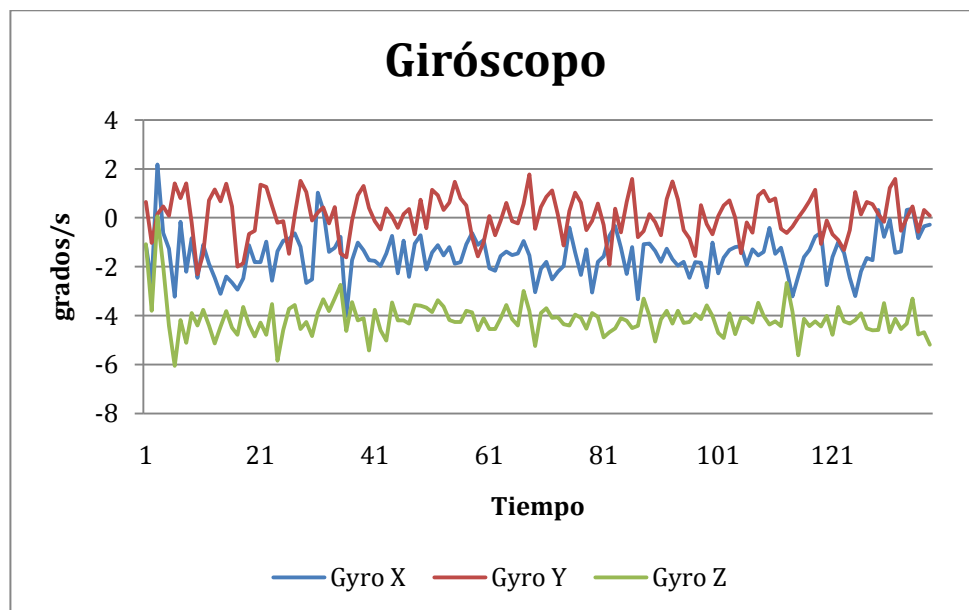


Fig. 50. Datos de la IMU (Gyro).

PRUEBA 3

En la siguiente prueba de trayectoria curva, se ha utilizado un ángulo de giro aún más grande que en la prueba anterior, realizando una trayectoria circular completa. En este caso, el odómetro del robot real captura el movimiento circular de forma correcta, un movimiento consistente en dar casi dos vueltas en círculo. La simulación no refleja este movimiento en círculo, dado que para el mismo tiempo de ejecución no se llega a completar toda la trayectoria, por lo que será necesario modificar las unidades de distancia en la simulación, para que se asemejen a las unidades de distancia reales.

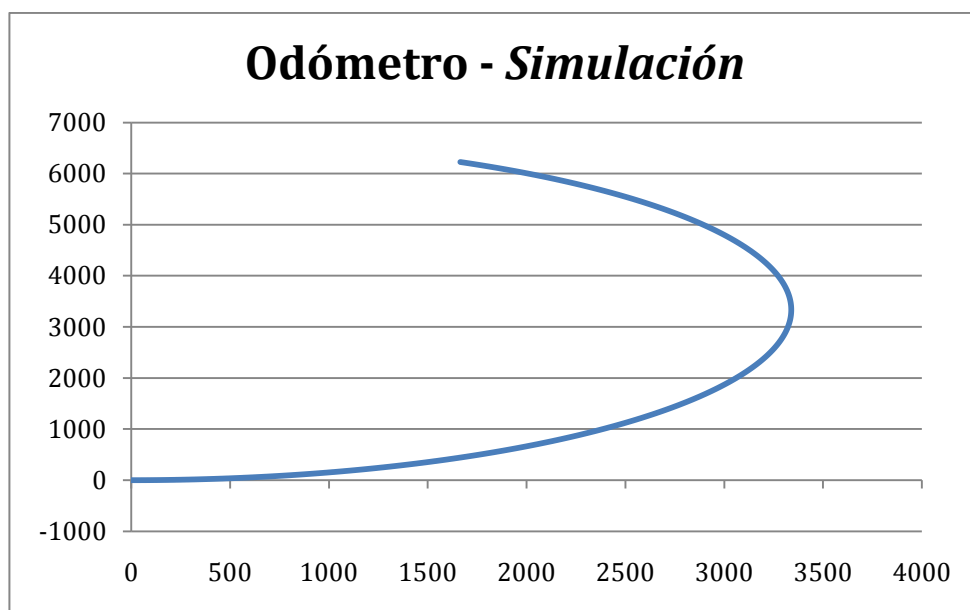


Fig. 51. Trayectoria simulada

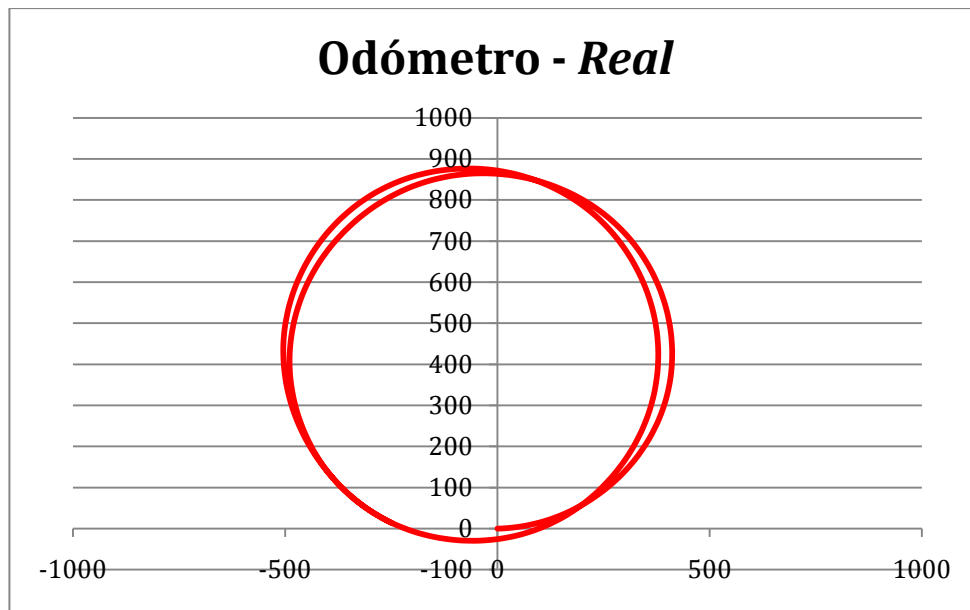


Fig. 52. Trayectoria real

Los datos capturados por la IMU durante esta prueba de trayectoria circular, arrojan los siguientes resultados.

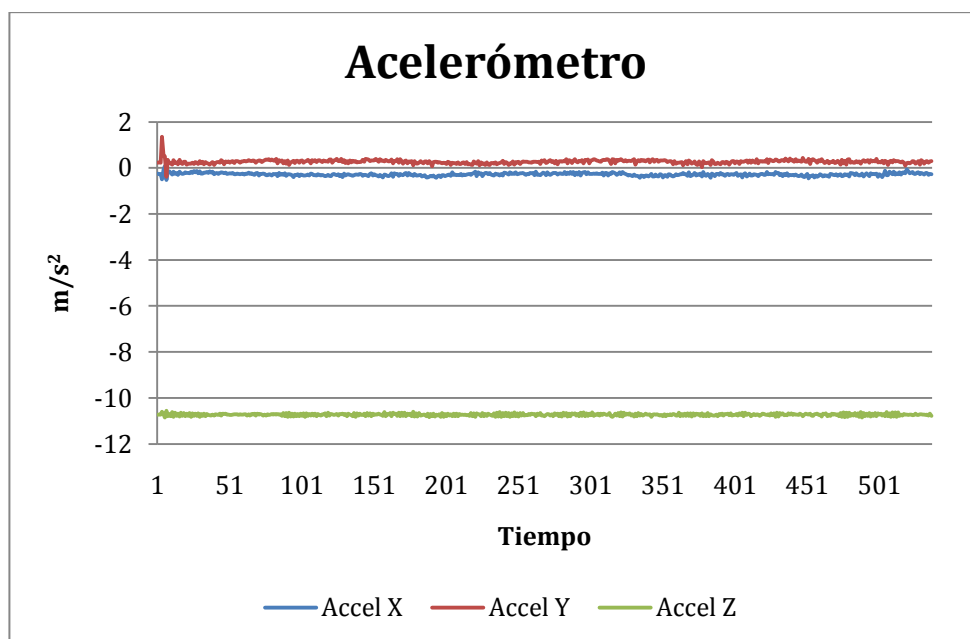


Fig. 53. Datos de la IMU (Aceleración)

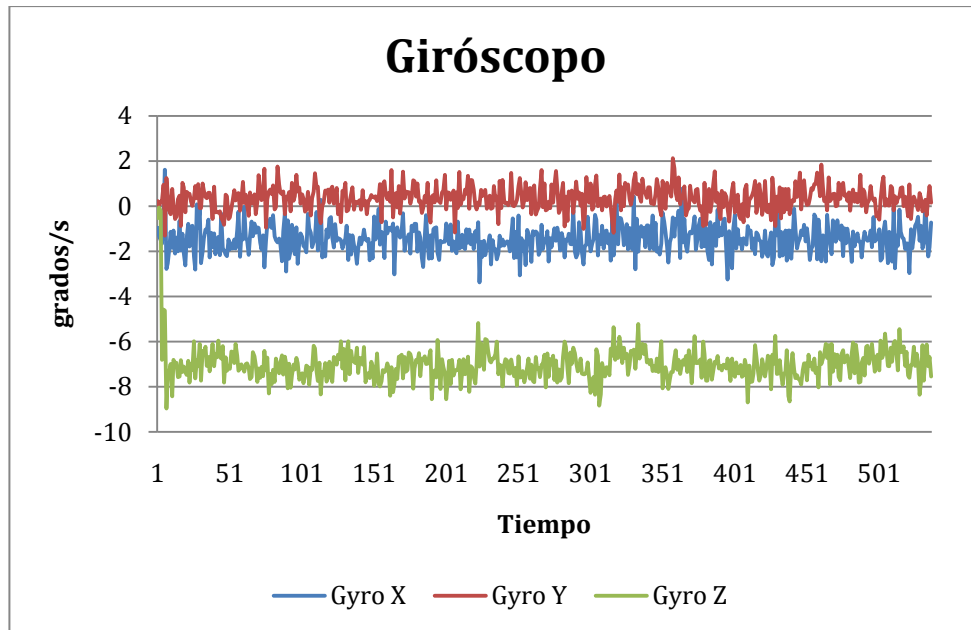


Fig. 54. Datos de la IMU (Gyro).

El acelerómetro registra una pequeña aceleración al inicio de la prueba, debido al arranque del robot para coger la velocidad necesaria. Después se puede apreciar que no se produce ninguna aceleración en ninguno de los tres ejes, pero se sigue observando mucho ruido, el cual impide reconocer aceleraciones muy pequeñas, ya que estas se encuentran dentro del rango sobre el cual fluctúa el ruido.

En cuanto el giróscopo, en esta ocasión se aprecia un claro giro sobre el eje Z de manera continua a lo largo de todo tiempo que dura la prueba. El giróscopo ha recogido datos de la trayectoria circular, los cuales reflejan un giro constante en torno a los 7 grados, aunque es imposible estimar con precisión el ángulo exacto, puesto que el ruido en los datos medidos es bastante grande.

PRUEBA 4

En la siguiente prueba de trayectoria curva, se ha utilizado un ángulo de giro aún más grande que en la prueba anterior, realizando una trayectoria circular completa pero más cerrada que en el caso anterior, de forma que se pueda comprobar si la IMU responde ante estos movimientos y giros.

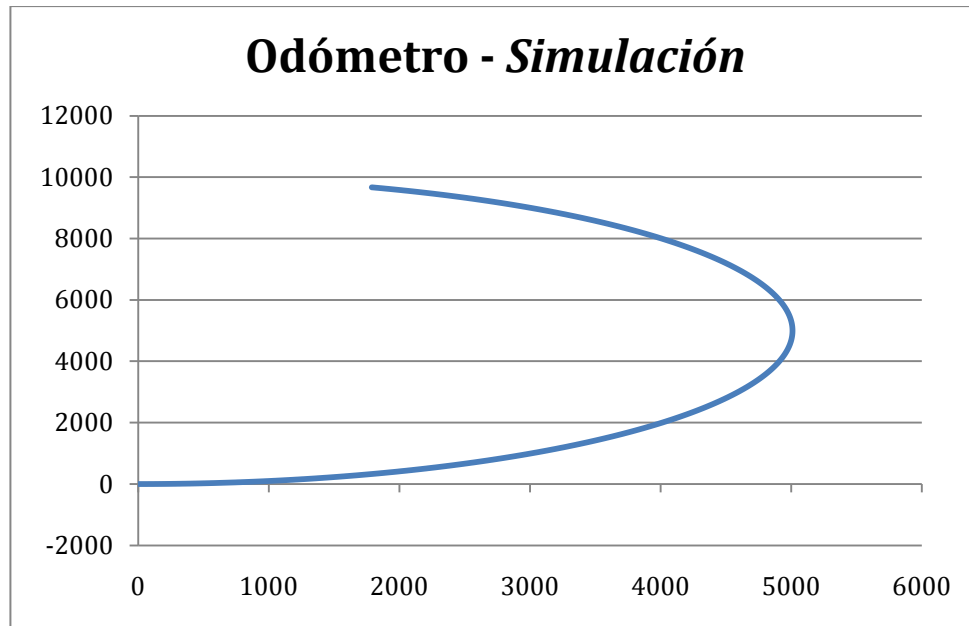


Fig. 55. Trayectoria simulada

La trayectoria de la simulación, al igual que sucede en la prueba anterior, no se asemeja a una circunferencia, puesto que las distancias no son equivalentes a las de la realidad y por tanto se necesitaría más tiempo para lograr realizar una trayectoria circular completa. Para solucionar este inconveniente hay que ajustar las unidades métricas.

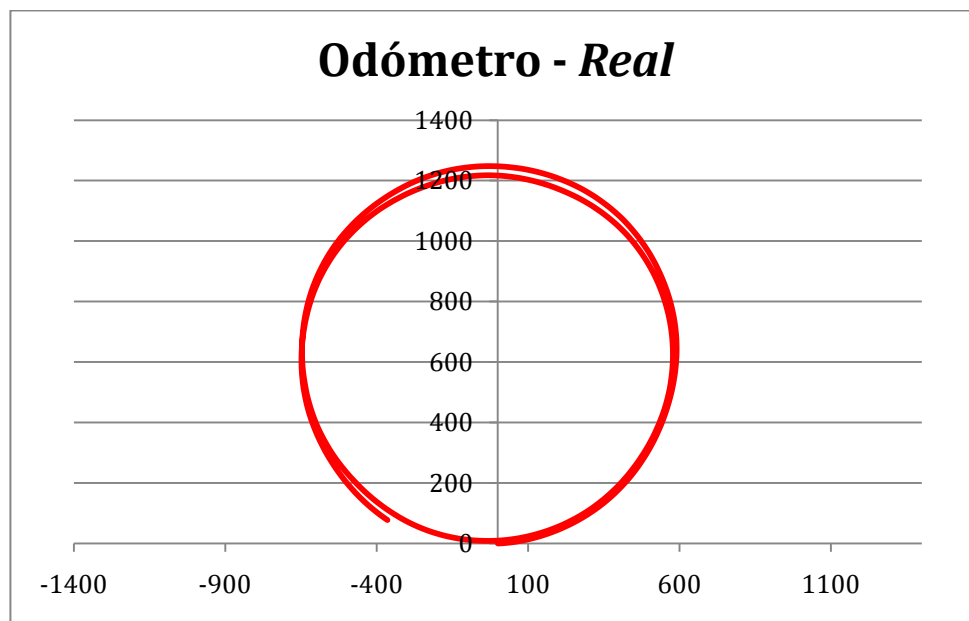


Fig. 56. Trayectoria real

El odómetro del robot real capta muy bien el recorrido realizado por el robot. Es posible que realizando movimientos suaves y continuos, el odómetro se comporte

mucho mejor que si se llevan a cabo giros bruscos, como los realizados en las primeras pruebas, cuyos resultados no eran nada satisfactorios.

Respecto al sensor inercial del robot, el acelerómetro continúa recogiendo la pequeña aceleración al inicio de la prueba, a pesar de que el ruido sigue siendo muy alto, hecho que se mantiene constante e independiente de las pruebas y que precisa de un reajuste.

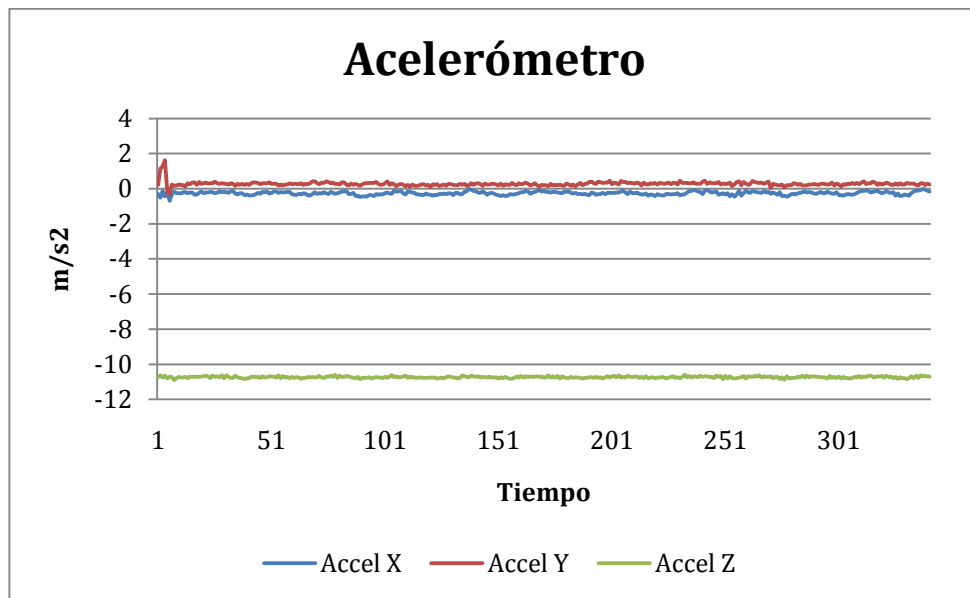


Fig. 57. Datos de la IMU (Acelerómetro).

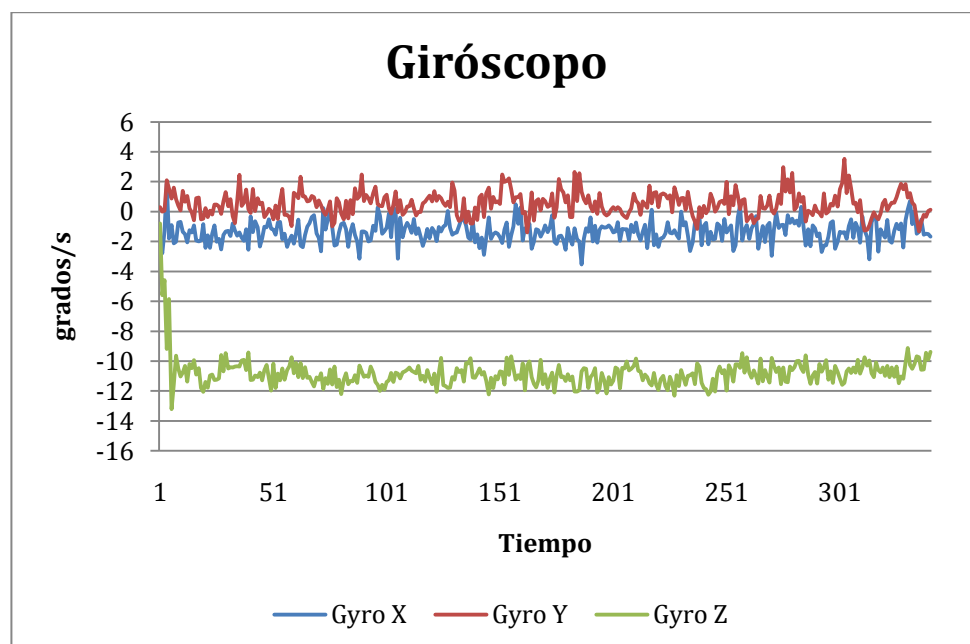


Fig. 58. Datos de la IMU (Gyro).

Al igual que en las pruebas anteriores de trayectorias curvas, el giróscopo detecta el movimiento de giro del robot, que en esta ocasión se produce durante todo el recorrido y con un valor de 11 grados/s aproximadamente, aunque el error que presentan los datos medidos vuelve a hacer casi imposible conocer el verdadero movimiento realizado por el robot a lo largo del tiempo de la prueba.

A falta de la realización de más pruebas, se puede concluir que el sensor odómetro (y/o su software de medición) del robot se comporta bien si los movimientos son suaves y continuos, pero que al realizar giros bruscos, se pierde prácticamente toda clase de fiabilidad en la trayectoria medida.

En cuanto al sensor inercial, el ruido que se obtiene en los datos es sumamente grande. Este ruido en ocasiones puede deberse a las vibraciones propias del robot cuando éste se encuentra en movimiento, pero de cualquier modo, supone un grave impedimento para su utilización, puesto que se ha demostrado que es incapaz de detectar cambios pequeños de movimientos angulares y de aceleración. Asimismo, tampoco se registran aceleraciones laterales durante los giros, hecho que puede no darse debido al giro de tipo diferencial que posee el robot.

Para concluir con la primera valoración sobre los experimentos realizados, se comenta a continuación el comportamiento de la capa de software desarrollada, la cual ha sido utilizada en el programa de control de las pruebas.

Durante todas las pruebas, los datos leídos a través de los dispositivos sensores se han ido almacenado en la estructura en forma de lista del middleware. De este modo no ha sido necesario tratar los datos en el momento de la lectura y ha sido posible utilizarlos más adelante, concretamente al final de las pruebas, en las que se vuelca toda la información de los sensores en un archivo de texto plano para su posterior interpretación y análisis. Esto supone una gran ventaja ya que si el sistema sobre el cual se está ejecutando el programa de control no es muy potente, entonces no podrá procesar todos los datos en tiempo real, pero sí que podrá procesarlos poco a poco según las necesidades ya que los datos no se pierden, si no que se almacenan en la lista.

En lo que respecta a los datos recogidos por los sensores y que se han ido almacenando en la estructura en forma de lista de capa de software intermedia, cabe destacar que cuando el robot se encuentra parado se detectan datos repetidos provenientes del sensor Odómetro y del sensor Láser, ya que al no cambiar de posición se procesan como datos repetidos y por tanto son descartados. Esto no sucede con los demás sensores, debido al ruido que poseen, el cual hace que siempre haya variaciones en los datos recogidos y por tanto se procesen como nuevos datos.

Por otro lado, se ha observado que con la configuración actual es posible recoger al menos 10 datos por segundo de cada uno de los sensores instalados en el robot Guardián. El sensor inercial es quizás el dispositivo que más rápido captura datos y posiblemente alguno de estos datos se pierda, ya que una tasa de lectura de 10 Hz puede no ser suficiente. Para las necesidades actuales, el número de datos recogidos

es en principio sí es suficiente, por lo que a priori no es necesario recurrir a la multiprogramación para evitar la pérdida de datos causada por la asincronía de los sensores.

Puesto que las pruebas realizadas han estado orientadas al análisis de los sensores y de la captura de los datos, el middleware no ha podido ser probado con más profundidad y por tanto las valoraciones sobre su funcionamiento no pueden ser más amplias.

CONCLUSIONES

CAPITULO

Este proyecto llega a su fin después de haber finalizado la experimentación y haber obtenido los datos necesarios para efectuar una valoración del sistema desarrollado, de forma que se pueda decidir si la solución obtenida es susceptible de ser utilizada en un futuro. En esta valoración se remarcan las ventajas e inconvenientes a la hora de utilizar una capa de software especialmente diseñada para aislar a los componentes de más alto nivel de las tareas de comunicación con el hardware de un robot y para proporcionarles diversos métodos de acceso a los datos capturados por los sensores del robot.

Al inicio de este proyecto se describían las cualidades y características de los robots autónomos y de la importancia que tiene la navegación en estos. La navegación se basa en el uso de dispositivos sensores que de alguna manera permitan conocer la posición o la variación en el movimiento. Por tanto, tal y como se explicaba en los primeros puntos de este documento, la fiabilidad de los datos proporcionados por los sensores es un punto crucial, de modo que la aplicación de sistemas de fusión de datos de sensores se vuelve casi imprescindible. Estos sistemas de fusión de datos de sensores, trabajan con los datos provenientes de múltiples dispositivos sensores, como IMU o GPS, con el objetivo de aumentar la precisión de los cálculos de la posición. El software desarrollado a lo largo de este proyecto, pretende ser un intermediario entre los sistemas de fusión de datos y los propios dispositivos sensores, de forma que las tareas de comunicación y obtención de datos a partir de los sensores, sean totalmente transparentes para estos sistemas de fusión de datos.

La principal ventaja de utilizar una capa intermedia de software entre los sistemas de fusión de datos de sensores y el hardware del robot, es sin duda el hecho de que los sistemas de fusión de datos se vuelven más independientes y portables al no tener que

implementar de manera específica la comunicación con los dispositivos sensores del robot. Únicamente estos sistemas han de solicitar los datos a la capa intermedia y ésta se los proporcionará, sin importar la manera en la que los datos son adquiridos. Esto significa que gracias a la inclusión de la capa intermedia desarrollada, pueden utilizarse los sistemas de fusión ya existentes, realizando mínimos cambios para su correcto funcionamiento.

Otra ventaja de utilizar una capa intermedia como la que se ha desarrollado en este proyecto, es la organización de los datos en una estructura de tipo lista. Gracias a esta organización, resulta muy sencillo trabajar con los datos. Además, el hecho de almacenar los datos en una estructura y no enviarlos directamente a los sistemas de fusión de datos, reduce notablemente los posibles problemas de sobrecarga que pueden aparecer en los sistemas de fusión de datos, cuando se intenta procesar muchos datos a la vez. De esta forma, es el propio sistema de fusión de datos el que solicita los datos cuando los necesite realmente.

En lo que respecta a los resultados obtenidos a partir de las diferentes pruebas realizadas durante la fase de experimentación, cabe destacar que el uso de una capa intermedia no afecta para nada en los resultados y de hecho resulta indetectable a simple vista, lo que quiere decir que no se produce ningún tipo de retraso por el hecho de que los datos se almacenen previamente en una estructura de tipo lista. Esto se debe en gran medida a que la frecuencia de lectura de Player es de 10Hz, por lo que la capa intermedia es considerablemente más rápida a la hora de almacenar los datos en la estructura, los cuales ocupan unos cuantos bytes.

Otro aspecto importante que arrojan los resultados de los experimentos, es la poca precisión que posee el sensor inercial o IMU. Los datos proporcionados por este sensor poseen un ruido muy elevado. Este ruido supone un error de $\pm 0.3 \text{ m/s}^2$ en los componentes del acelerómetro, por lo que es imposible detectar el movimiento del robot a velocidades muy bajas, hasta el punto de no saber si quiera si el robot se encuentra parado. El giróscopo se comporta mejor que el acelerómetro, pero también posee un ruido considerable, cercano a ± 1.5 grados, lo que a la larga puede ocasionar la pérdida total de la orientación del robot. Estos datos inducen a pensar en la posibilidad de que el dispositivo de medición inercial esté descalibrado o estropeado, ya que si el error fuese menor, bastaría con una corrección vía software, como suele hacerse en estos casos.

Por su parte, el sensor odómetro encargado de registrar las vueltas que da cada par de ruedas (ruedas derechas y ruedas izquierdas), funciona bastante bien si la trayectoria seguida por el robot es una trayectoria suave, con curvas sencillas y sin realizar cambios bruscos en la dirección. En cambio si la trayectoria posee giros rápidos (por ejemplo girar 90 grados sobre sí mismo), el odómetro no consigue registrar ese cambio de manera satisfactoria, perdiendo la orientación. Queda claro que la utilización del sensor odómetro como única fuente de datos para la navegación del robot, es inviable debido a que no es un sensor fiable en lo que a movimientos giratorios se refiere.

Para concluir con las observaciones y las valoraciones sobre el proyecto, cabe destacar ciertos aspectos sobre la capa intermedia de software desarrollada para el control del robot Guardián. La implementación de dicha capa de software se apoya en las librerías proporcionadas por la plataforma Player/Stage y por tanto, hereda los problemas e inconvenientes de esta. Concretamente el problema de la asincronía, explicado en el apartado 5.1.1, es un problema que afecta directamente al sistema de control del robot. Para minimizar este problema se ha planteado la aplicación de técnicas de paralelización, de forma que la velocidad de lectura de los datos de los sensores, pueda ajustarse a cada uno de los diferentes tipos de sensor que incorpora el robot Guardián. Así pues, la lectura de cada sensor se realizaría de forma independiente.

La solución consistente en implementar varios hilos de ejecución para realizar la lectura de cada uno de los sensores de manera independiente, ha sido finalmente descartada (por el momento), debido a que las pruebas y los análisis realizados indican que no es necesario aplicar estos métodos para minimizar la asincronía. Player realiza las lecturas de los datos de los sensores a una velocidad de 10Hz por defecto. Velocidad suficiente para el uso que se le va a dar al robot Guardián en la actualidad y a corto/medio plazo. El único problema derivado de la asincronía entre los sensores que todavía puede persistir, es la repetición de datos, causado por los sensores más lentos como el GPS. Gracias al uso de la capa intermedia desarrollada, la cual realiza una serie de comparaciones, se puede evitar la repetición de datos y con ello una disminución de los cálculos a realizar.

En cuanto a las líneas de trabajo a seguir en un futuro a partir de este proyecto, probablemente pasen por la construcción de un sistema de control más complejo para el robot Guardián, ya que el sistema implementado en este proyecto es un sistema de control sencillo y de primera toma de contacto. Así mismo, se integrarán los algoritmos de fusión de datos existentes con el sistema desarrollado, haciendo uso de la capa de software intermedia, de forma que la construcción de un robot autónomo comience a ser una realidad. Realmente este proyecto sirve como documentación para futuros proyectos, proporcionando un análisis sobre las características del robot Guardián y del software relacionado, además de instaurar unas sencillas bases para la construcción de un sistema de control mucho más elaborado y con más posibilidades.

BIBLIOGRAFÍA

CAPITULO

En este capítulo se recogen las referencias a otros documentos o fuentes de información que han sido utilizadas para la elaboración de este proyecto. En ciertas ocasiones es posible que se haga referencia a una dirección Web de internet, sitio en el que se podrá encontrar toda la información detallada correspondiente a esa referencia. La bibliografía se encuentra enumerada por orden de aparición en el documento para una mayor comodidad.

[1] Cox, I.J., Wilfong, G.T. *"Autonomous robot vehicles"* - Springer-Verlag New York, Inc. 1990

[2] Gallington, R.W., Berman, H., Entzminger, J., Francis, M.S, Palmore, P., Stratakes, J. *"Unmanned aerial vehicles"* - Future aeronautical and space systems (A97-26201 06-31), Reston, VA, American Institute of Aeronautics and Astronautics, Inc. (Progress in Astronautics and Aeronautics. Vol. 172), 1997, p. 251-295

[3] Maksarov, D., Durrant-Whyte, H. *"Mobile vehicle navigation in unknown environments: a multiple hypothesis approach."* - IEE Proceedings. Control Theory and Applications. Vol. 142, no. 4, pp. 385-400. July 1995

[4] Hall, D.L., Llinas, J. *"An introduction to multisensor data fusion"* - Proceedings of the IEEE Volume 85, Issue 1, Jan. 1997 Page(s):6 - 23

[5] Documentación de Player/Stage (<http://playerstage.sourceforge.net>)

- [6] Gerkey, B.P., Vaughan, R.T., Howard, A. *"The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems"* - Proceedings of the International Conference on Advanced Robotics (ICAR 2003). Pages 317-323. 2003
- [7] Groves, P.D. *"Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems"* - Artech House, 2008
- [8] Guerrero, J.L., Garcia, J., Molina, J.M. *"Multi-agent Data Fusion Architecture proposal for Obtaining an Integrated Navigated Solution on UAV's"* - International Symposium on Distributed Computing and Artificial Intelligence (DCAI 2009). 2009
- [9] González, A.R., Rubio, J.F. *"Integración gps/ins: conceptos y experimentos"* – Universidad Politécnica de Cataluña
- [10] Brown, R.G. *"Receiver Autonomous Integrity Monitoring"* - Global Positioning System: Theory and Applications, Volume II, AIAA, Washington D.C. 1996
- [11] Dorf, R., Bishop, R.H. *"Sistemas lazo abierto"* - Sistemas de control moderno, Prentice Hall, 2005
- [12] Dorf, R., Bishop, R.H. *"Sistemas lazo cerrado"* - Sistemas de control moderno, Prentice Hall, 2005
- [13] Postel, J. *"RFC 793: Transmission Control Protocol"* - September 1981
- [14] Espinosa, F., Salazar, M., Valdes, F., Bocos, A. *"Communication architecture based on Player/Stage and sockets for cooperative guidance of robotic units"* - 16th Mediterranean Conference on Control and Automation. 2008
- [15] Bates, M.P. *"PIC Microcontrollers: An Introduction to Microelectronics"* - 2nd edition. Newnes. 2004
- [16] Di, L., Yu, G.E., Chen, N., Min, M., Wang, H. *"Support of Asynchrony in Sensor Web"* - Center for Spatial Information Science and Systems 07/13/2007
- [17] Andrews, G.R. *"Foundations of Multithreaded, Parallel, and Distributed Programming"* - Addison-Wesley 1999
- [18] Matthies, L., Kelly, A., Litwin, T., Tharp, G. *"Obstacle detection for unmanned ground vehicles: a progress report"* - Proceedings of the Intelligent Vehicles '95 Symposium. Pages 66-71. 1995

ANEXOS

CAPITULO

En este último capítulo de la memoria del proyecto, se detallan los anexos al documento principal. Estos anexos recogen en detalle diversas explicaciones sobre elementos relacionados con el proyecto, pero que quedan fuera del concepto general del mismo. Tratan puntos relacionados con la gestión del proyecto en términos de planificación y costes, manuales y guías de instalación del software utilizado durante el desarrollo de este proyecto, etc.

Los anexos son independientes entre sí y no comparten contenido alguno, salvo ciertas excepciones que son mera coincidencia. Cada anexo se identifica por una letra del alfabeto, empezando por la letra A.

GESTIÓN DEL PROYECTO

ANEXO

En este anexo se recoge la información relacionada con la gestión del proyecto, concretamente se describe la planificación que se ha seguido y los costes asociados a cada uno de los diferentes aspectos de los que se compone el proyecto.

Al inicio de todo proyecto, es necesario establecer un plan de trabajo en el que hay que fijar unos límites de tiempo para la realización de cada una de las tareas de las que se compone. De este modo, es más sencillo controlar el estado del proyecto en todo momento durante su ciclo de vida, haciendo posible realizar ajustes si la situación lo requiere. Generalmente es muy difícil cumplir con los tiempos preestablecidos para cada tarea, pero llevar una planificación actualizada minimiza considerablemente el impacto de los retrasos.

Respecto a la gestión económica, para un proyecto pequeño como este, es conveniente generar informes sobre los gastos y las inversiones, con el fin de realizar un presupuesto inicial y ajustarse lo máximo posible a las condiciones económicas de las que se dispone. Por ello, en este apartado se detallan los gastos en inversión de materiales y de recursos humanos.

La planificación de un proyecto constituye además una importante fuente de información para futuros trabajos, puesto que analizando los datos referentes a la planificación, es posible determinar los errores en las estimaciones, tanto de tiempo como de costes, minimizando así los posibles problemas derivados de las malas predicciones en los futuros proyectos.

PLANIFICACIÓN DE LAS TAREAS

El proyecto se divide en un total de diecisiete tareas, a cada una de las cuales se le ha asignado una duración estimada en la que van a llevarse a cabo. Inicialmente se realiza una estimación de los tiempos que puede requerir cada tarea hasta su consecución, pero a medida que el proyecto avanza, es necesario realizar ajustes en la planificación, debido a los problemas e inconvenientes que inevitablemente surgen en todos los proyectos.

A continuación se presenta la planificación establecida al inicio de este proyecto, en la cual se pueden apreciar cada una de las etapas en las que se encuentra dividido (Fig. 43), acompañado del diagrama de Gantt correspondiente (Fig. 44), el cual permite tener una visión rápida y global de la planificación del proyecto.

Nº	Título	Duración	Fecha Inicio	Fecha Fin
0 ▼ 📁	PFC		01/02/09	30/12/09
1	Generación de documentos	330 días	01/02/09	27/12/09
2 ▼	Estudio de Player/Stage		02/02/09	04/04/09
3	Estudio documentación	20 días	02/02/09	21/02/09
4	Instalación Player/Stage	10 días	02/03/09	11/03/09
5	Pruebas simulador Stage	10 días	16/03/09	04/04/09
6 ▼	Estudio del robot Guardian		05/04/09	30/05/09
7	Estudio documentación	30 días	05/04/09	04/05/09
8	Configuración del hardware	3 días	05/05/09	07/05/09
9	Instalación del software	5 días	05/05/09	09/05/09
10	Conexión PC-Robot	6 días	05/05/09	10/05/09
11	Pruebas robot Guardian	20 días	11/05/09	30/05/09
12 ▼	Desarrollo		02/06/09	04/10/09
13	Análisis de las necesidades	15 días	02/06/09	16/06/09
14	Diseño del Middleware	20 días	17/06/09	06/07/09
15	Implementación	60 días	07/07/09	04/09/09
16	Pruebas con el middleware	30 días	05/09/09	04/10/09
17	Elaboración de la memoria	80 días	05/10/09	23/12/09

Fig. 59. Planificación de las tareas.

El proyecto se ha planificado de forma que son tres las etapas por las que ha de pasar hasta llegar a su finalización. Cada una de las etapas se compone a su vez de tareas, las cuales tienen un tiempo de realización estimado en días de trabajo. Para este proyecto, cada día de trabajo supone seis horas. Estas etapas corresponden a:

- ⌘ Estudio de Player/Stage.
- ⌘ Estudio del robot Guardián.
- ⌘ Desarrollo.

En la primera etapa, se realiza un estudio de la plataforma Player/Stage, así como se lleva a cabo la instalación del software necesario para desarrollar en este entorno. Esta ha sido la primera etapa del proyecto debido a que el robot Guardián proporcionado por Robotnik Automation S.L. no estaba disponible al comienzo del proyecto.

La siguiente etapa consiste en estudiar y poner en marcha el robot Guardián, un robot real orientado como un UGV con múltiples dispositivos sensores. En esta fase o etapa, se estudian los manuales del robot para comprender todos los conceptos y se configura el hardware y el software para poder iniciar el desarrollo de programas de control.

La tercera y última etapa del proyecto, está dedicada al desarrollo propiamente dicho. En esta fase se analizan las necesidades planteadas al inicio del proyecto y se diseña una solución que permita cubrir esas necesidades. Esta solución se implementa con la ayuda de la plataforma Player/Stage y por último se realizan las pruebas pertinentes para asegurar el correcto funcionamiento.

Fuera de estas etapas, es necesario elaborar una memoria detallada que describa todos los aspectos del proyecto. Para la elaboración de esta memoria, durante todas las etapas descritas con anterioridad, se generan documentos descriptivos que finalmente conformaran la memoria final.

A continuación se muestra el diagrama de Gantt (Fig. 59) asociado a la planificación del proyecto. Por motivos de claridad no se muestran los nombres de las tareas, si no que se representan por sus números identificativos. La relación entre el número y el nombre de una tarea se encuentra en la Fig. 58.

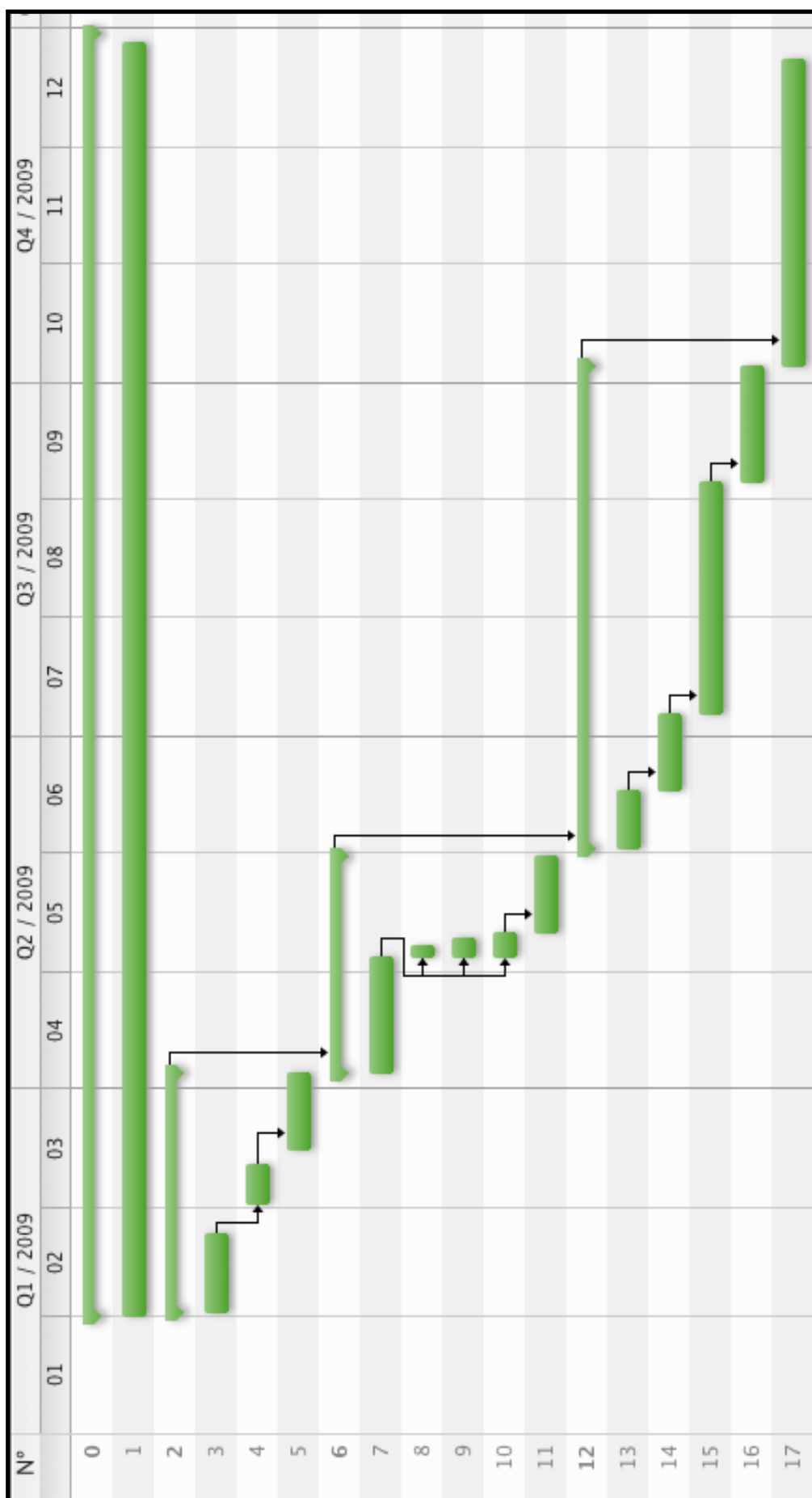


Fig. 60. Diagrama de Gantt para la planificación del proyecto.

GESTIÓN ECONÓMICA DEL PROYECTO

Otro aspecto importante en la gestión de un proyecto es sin duda el manejo de los aspectos económicos del proyecto, tales como la inversión en materiales o la contratación de trabajadores. En definitiva se trata de controlar el gasto final. Para la realización de esta labor, generalmente se utilizan herramientas específicas, pero dadas las características de este proyecto, con una estimación es suficiente.

En este apartado se describen los presupuestos asignados al inicio de este proyecto, diferenciando cada uno de los aspectos presupuestados.

Recursos Hardware/Software

El primer aspecto importante en el que hay que fijarse al inicio de un proyecto, es el material que va a ser necesario para poder llevar a cabo el proyecto. El material tiene que cubrir las necesidades tanto de hardware como de software del proyecto, por lo que hay que asegurarse de que los materiales adquiridos sean lo suficientemente modernos y capaces, ya que de lo contrario, pueden aparecer muchos problemas derivados de las malas prestaciones, lo que repercute en el presupuesto del proyecto.

Para la realización de este proyecto, se requieren diversos materiales tanto de hardware como de software. A continuación se listan las características mínimas e imprescindibles que han de tener los diferentes componentes:

⌘ Ordenador personal (PC):

- Procesador Intel Pentium4 @ 3GHz o AMD similar.
- 1 GB de memoria RAM.
- Tarjeta gráfica con 64MB de memoria.
- Disco de duro de 100GB.
- Joystick estándar.

⌘ Robot con múltiples dispositivos sensores (UGV):

- Sensor IMU.
- Sensor GPS.
- Sensor Láser.

⌘ Software:

- Sistema operativo Ubuntu 9.10 de tipo Linux.
- Sistema operativo Xubuntu8.01 adaptado a las características del robot.
- Herramientas Player/Stage.
- Microsoft Office 2007 (Word y PowerPoint).

Recursos humanos

Respecto a los recursos destinados a los trabajadores, primero hay que identificar el personal que va a ser necesario para realizar el proyecto de acuerdo a la planificación presentada con anterioridad. En este caso, sólo una persona, Tomás Rebollo, es la encargada de desarrollar todo el proyecto, por lo que esta persona asume todos los roles del proyecto: jefe de proyecto, diseñador y analista programador.

El coste asociado al salario del personal de trabajo, que en esta caso corresponde únicamente al salario de una persona, Tomás Rebollo, se obtiene a partir de una estimación del número de horas que esta persona va trabajar en el proyecto. Esta estimación se basa en la planificación del proyecto, descrita en el punto anterior, teniendo en cuenta que cada día de trabajo suponen un total de seis horas.

DÍAS TRABAJADOS	HORAS/DÍA	COSTE POR HORA	COSTE TOTAL EN EUROS
330 días	6 h/día	20 €/h	39600 €

Tabla 8. Desglose del gasto en RR.HH.

Además de los gastos en recursos humanos y el equipo, tanto hardware como software, durante todo el proyecto se utilizan materiales consumibles, como es el papel para fotocopias, herramientas de trabajo (destornilladores, llaves, etc.), cables de conexión, etc., por lo que al presupuesto hay que añadir una pequeña cantidad para gastos imprevistos. Dadas las características de este proyecto, con una cantidad de 200 € aproximadamente, todos estos aspectos quedan cubiertos.

A continuación se muestra una tabla que contiene el desglose de todos los elementos del presupuesto y el gasto total del proyecto.

CONCEPTO	COSTE TOTAL
RR.HH.	39600 €
Hardware	19500€
Software	0 €
Imprevistos	200 €
TOTAL	59300 €

Tabla 9. Presupuesto del proyecto.

GUÍA DE INSTALACIÓN DE

ANEXO

El conjunto de herramientas de Player/Stage conforman una plataforma de código abierto y libre, destinada a su uso en sistemas robóticos con múltiples dispositivos sensores, tal y como se ha descrito en el capítulo 3 de este documento. Para comenzar a utilizar este conjunto de herramientas, que incluyen un serie de librerías de apoyo para la programación de programas de control y un completo entorno de simulación en el que realizar diversas pruebas, es necesario instalar los componentes software y configurarlos correctamente.

En este anexo se describen los pasos necesarios que hay que seguir para conseguir instalar las herramientas de Player/Stage en un ordenador personal normal y obtener un entorno de desarrollo de programas de control para el robot completamente funcional.

PASO 1

Las herramientas de Player/Stage funcionan bajo sistemas de tipo Unix como Linux. Actualmente (a fecha de Diciembre de 2009) se está portando Player/Stage al sistema operativo Windows, de forma que las posibilidades de uso sean mucho mayores. Sin embargo, lo más fácil y seguro es utilizar Player/Stage sobre una de las muchas distribuciones del sistema operativo Linux. En el caso de esta guía, se utiliza la distribución de Ubuntu en su versión 9.10.

Si no se dispone de un PC con sistema operativo Ubuntu, el primer paso es descargar la última versión desde la página oficial de Ubuntu (<http://www.ubuntu.com>), y realizar la instalación del sistema operativo de la forma habitual (siguiendo las indicaciones que el instalador va mostrando). Una vez instalado y configurado el nuevo sistema, proceder con el paso 2.

Cabe destacar que se puede el sistema se puede instalar sobre una máquina virtual si se prefiere, para lo cual hay que seguir exactamente los mismos pasos descritos en esta guía.

PASO 2

El siguiente paso a la instalación del sistema operativo Ubuntu, consiste en instalar Player/Stage en el sistema. Para empezar con la instalación, es imprescindible disponer de los archivos necesarios que contienen las herramientas de Player/Stage. Estos archivos se pueden obtener directamente desde la página Web de la plataforma Player/Stage (<http://playerstage.sourceforge.net>). Player/Stage está compuesto por dos herramientas separadas, por lo que han de descargarse los ficheros

correspondientes a ambas herramientas. Concretamente los archivos descargados para la realización de esta guía son:

- ⌘ Player-3.0.0.tar.gz
- ⌘ Stage-3.2.2-Source.tar.gz

Los ficheros corresponden a las versiones más actualizadas de Player y Stage, que a fecha de Diciembre de 2009 son la versión 3.0.0 de Player y la versión 3.2.2 de Stage.

Antes de proceder con la instalación de ambas herramientas, hay que asegurarse de que el sistema Ubuntu tenga instalados y configurados los paquetes y librerías necesarios. A continuación se listan los paquetes requeridos por Player y por Stage:

- ⌘ cmake
- ⌘ build-essential
- ⌘ libgtk2.0-dev
- ⌘ libfltk1.1-dev
- ⌘ libltdl7
- ⌘ libltdl-dev
- ⌘ freeglut3-dev

Para instalar los paquetes listados arriba, puede utilizarse la herramienta de gestión de paquetes Synaptic que trae Ubuntu, o bien se pueden instalar mediante la ejecución de un comando en un terminal o consola. Para este segundo caso, bastará con abrir un terminal y ejecutar el siguiente comando:

```
$ sudo apt-get install build-essential cmake libgtk2.0-dev libfltk1.1-dev libltdl7 libltdl-dev freeglut3-dev
```

Una vez se dispone de los archivos necesarios que contienen las herramientas de Player/Stage, el siguiente paso es proceder con la instalación. En esta guía se han guardado los archivos descargados desde la página de Player/Stage en una carpeta llamada 'playerstage' dentro de la carpeta 'home' del usuario.

PASO 3

Primero se debe de instalar Player, ya que Stage depende en cierto modo de las librerías de Player. Para hacerlo, se abre un terminal y se ejecutan los siguientes comandos, teniendo en cuenta la carpeta donde se han guardado los archivos descargados previamente y de las versiones de los mismos.

```
$ cd playerstage
$ tar -zxvf player-3.0.0.tar.gz
$ cd player-3.0.0
$ mkdir build
$ cd build
$ cmake ../
$ make
$ sudo make install
```

Al finalizar la ejecución de los comandos anteriores, Player queda definitivamente instalado en el sistema. Las librerías de Player se instalan en la ruta por defecto, que es `'/usr/local'`. Para comprobar que todo funciona, basta con ejecutar en un terminal la siguiente sentencia:

```
$ player
```

Con lo que debería parecer algo parecido a la siguiente imagen:

```

Archivo  Editar  Ver  Terminal  Ayuda
~$ player
Registering driver
Player v.3.0.0
USAGE: player [options] [<configfile>]

Where [options] can be:
-h          : print this message.
-d <level>  : debug message level (0 = none, 1 = default, 9 = all).
-p <port>   : port where Player will listen. Default: 6665
-q          : quiet mode: minimizes the console output on startup.
-l <logfile> : log player output to the specified file
<configfile> : load the the indicated config file

The following 91 drivers were compiled into Player:

AioToSonar accel_calib acts amcl amtecpowercube aadv bitlogic
bumper2laser bumper_safe cameracompress camerauncompress camerauvc
camerav4l camerav4l2 camfilter canonvcc4 clodbuster cmucam2 cmvision
create deadstop dummy epuck erl erratic fakelocalize festival
flockofbirds garminnmea globalize goto insideM300 iwspy kartowriter
khepera laserbar laserbarcode lasercspace lasercutter
laserposeinterpolator laserptzcloud laserrescan lasersafe
lasertoranger linuxjoystick linuxwifi localbb mapcspace mapfile
mapscale mica2 microstrain3dmg motionmind mricp nomad_driver obot p2os
passthrough pbs03jn ptu46 rangertolaser readlog relay rflex roboteq
robotracker roomba rs4leuze rt3xxx segwayrmp400 serialstream serio
sickLDMRS sicklms200 sicklms400 sicknav200 sickpls sickrfi341 sick3000
skyetekM1 snd sonartoranger sonyevide30 sphere tcpstream vec2map vfh
vmapi wavefront wbr914 writelog

~$

```

Fig. 61. Salida del terminal al ejecutar el comando 'player'.

Una vez instalado Player, se procede con la instalación de Stage, el entorno de simulación. Para ello hay que seguir prácticamente los mismos pasos que en la instalación de Player. Esto es:

```

$ cd playerstage
$ tar -zxvf Stage-3.2.2-Source.tar.gz
$ cd Stage-3.2.2-Source
$ mkdir build
$ cd build
$ cmake ../
$ make
$ sudo make install

```

Una vez ejecutados los comandos, Stage debería de estar instalado de forma correcta. Para que Stage sea utilizable junto con Player, es necesario decirle al sistema dónde se encuentran las librerías de Stage, que por defecto se instalan en la carpeta del sistema '/usr/local/lib'. Para ello, basta con ejecutar la siguiente sentencia cada vez que se inicie la sesión:

```
$ export LD_LIBRARY_PATH=/usr/local/lib
```

El comando anterior sirve para exportar las librerías de usuario y que el sistema las encuentre, pero habrá que hacerlo cada vez que iniciemos la sesión y abramos un nuevo terminal. Por eso, lo mejor es incluir esta sentencia en el fichero '.bashrc' del usuario, que se encuentra en la carpeta 'home' del usuario. Para ello, basta con abrir un editor de texto y al final del archivo '.bashrc' añadimos la sentencia de exportar anterior. Tener en cuenta que los archivos precedidos de un '.' son archivos ocultos y por tanto es posible que al intentar abrirlo desde un editor de texto gráfico, de la sensación de que no existe. Siempre se puede ejecutar el editor de texto desde un terminal, a través de la siguiente sentencia:

```
$ gedit .bashrc
```

Finalmente se comprueba que Stage está funcionando correctamente junto con Player, ejecutando en un terminal (partiendo de la carpeta home del usuario) la siguiente sentencia:

```
$ player playerstage/Stage-3.2.2-Source/worlds/simple.cfg
```

Al ejecutar la sentencia anterior, debería de aparecer una ventana con el entrono de simulación proporcionado por Stage, además de que en el terminal se muestra la información relativa a la ejecución de Player.

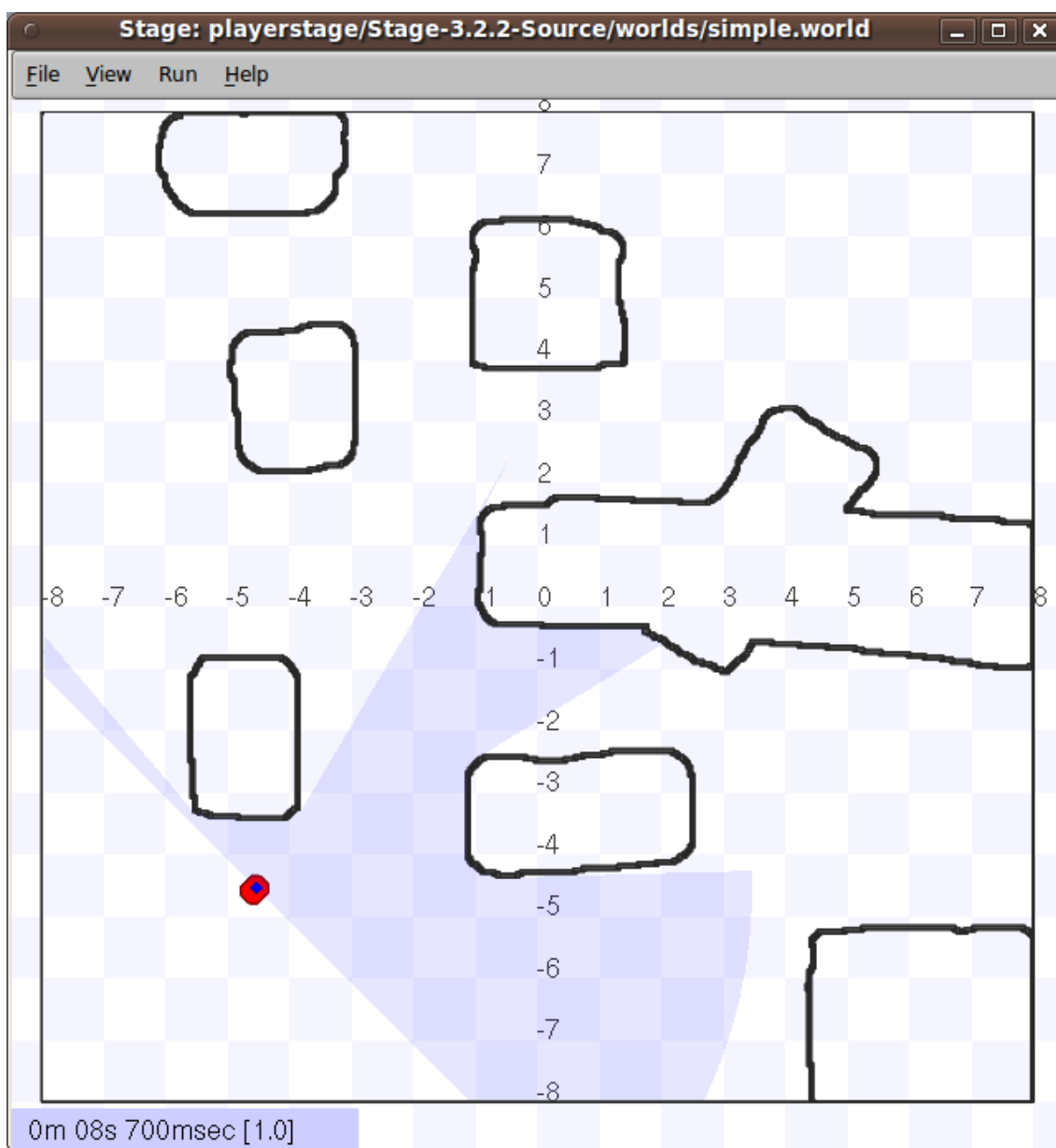
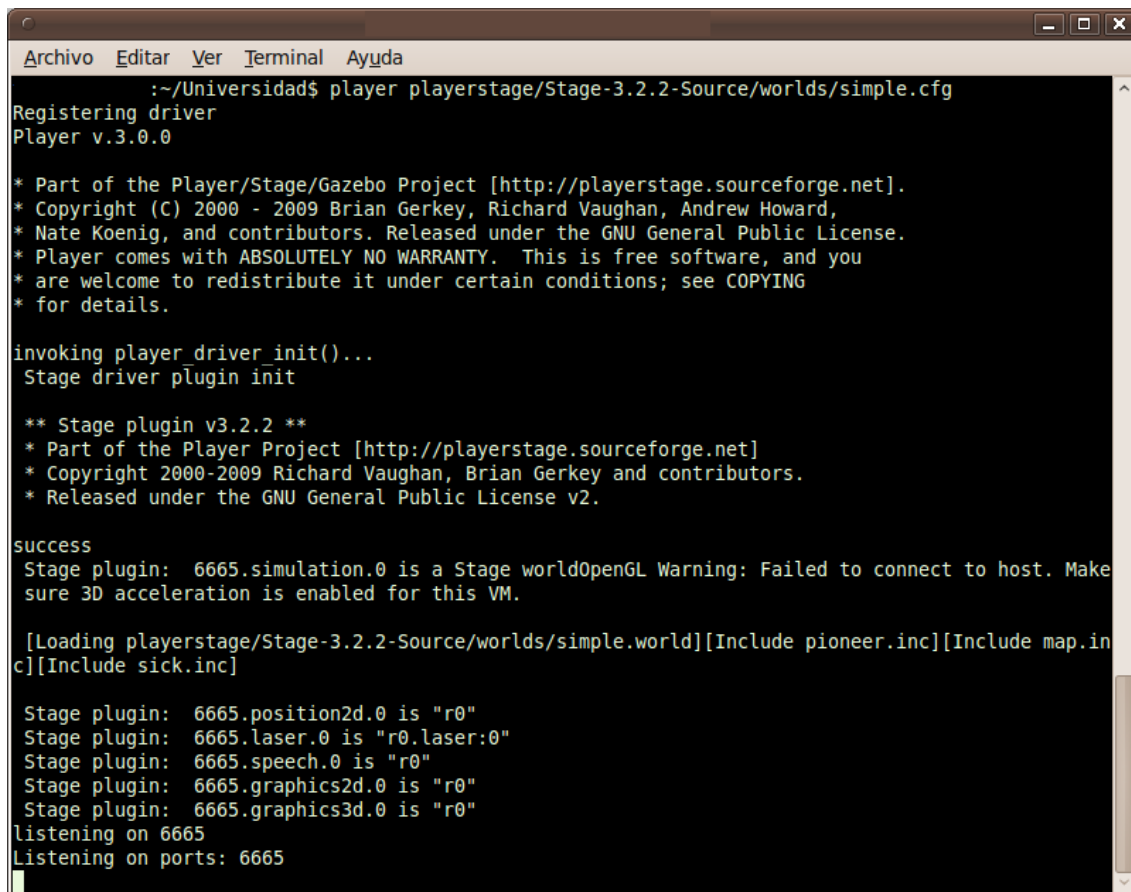


Fig. 62. Ventana del entorno de simulación Stage.



```
~/Universidad$ player playerstage/Stage-3.2.2-Source/worlds/simple.cfg
Registering driver
Player v.3.0.0

* Part of the Player/Stage/Gazebo Project [http://playerstage.sourceforge.net].
* Copyright (C) 2000 - 2009 Brian Gerkey, Richard Vaughan, Andrew Howard,
* Nate Koenig, and contributors. Released under the GNU General Public License.
* Player comes with ABSOLUTELY NO WARRANTY. This is free software, and you
* are welcome to redistribute it under certain conditions; see COPYING
* for details.

invoking player driver init()...
Stage driver plugin init

** Stage plugin v3.2.2 **
* Part of the Player Project [http://playerstage.sourceforge.net]
* Copyright 2000-2009 Richard Vaughan, Brian Gerkey and contributors.
* Released under the GNU General Public License v2.

success
Stage plugin: 6665.simulation.0 is a Stage worldOpenGL Warning: Failed to connect to host. Make
sure 3D acceleration is enabled for this VM.

[Loading playerstage/Stage-3.2.2-Source/worlds/simple.world][Include pioneer.inc][Include map.in
c][Include sick.inc]

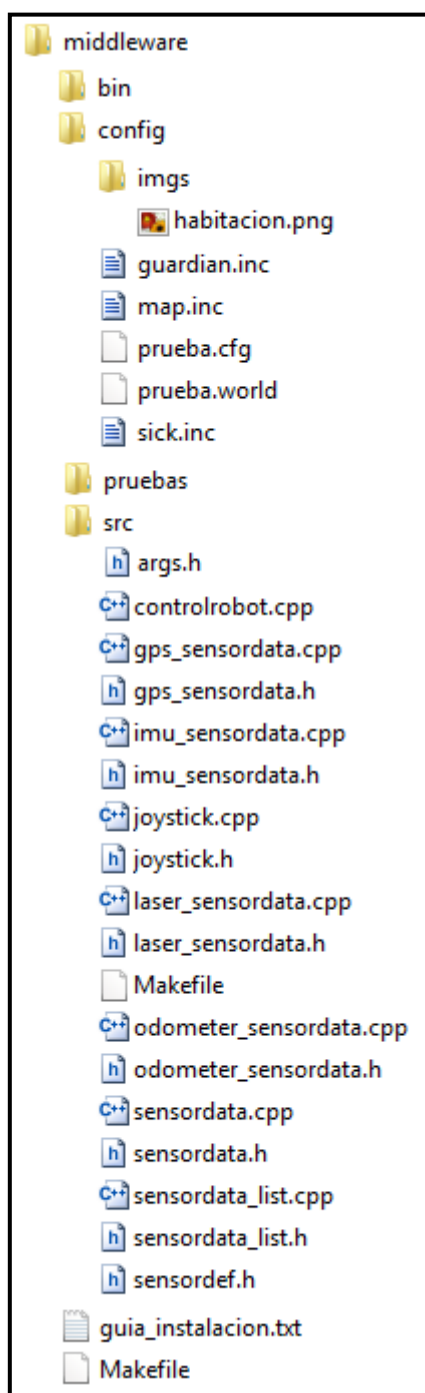
Stage plugin: 6665.position2d.0 is "r0"
Stage plugin: 6665.laser.0 is "r0.laser:0"
Stage plugin: 6665.speech.0 is "r0"
Stage plugin: 6665.graphics2d.0 is "r0"
Stage plugin: 6665.graphics3d.0 is "r0"
listening on 6665
Listening on ports: 6665
```

Fig. 63. Salida del terminal durante la ejecución de Player y Stage.

MANUAL DE USUARIO DEL MIDDLEWARE

ANEXO

En este anexo se describen los pasos para utilizar de manera sencilla la capa de software desarrollada. De este modo, cualquier persona puede iniciar un nuevo proyecto apoyándose en esta capa de software y así poder desarrollar programas de control más complejos y elaborados.



Lo primero es hacer un repaso de los archivos que componen esta capa intermedia de software, llamada middleware, así como mostrar su organización dentro de la carpeta del proyecto.

En la imagen de la izquierda puede verse el esquema de organización de los archivos que componen el proyecto, cuya raíz es la carpeta 'middleware'.

En la subcarpeta 'bin' se almacenan los ejecutables, una vez compilada la aplicación. En este caso únicamente se producirá un ejecutable, el cual se llamará 'controlrobot'.

La siguiente subcarpeta, llamada 'config', contiene los archivos de configuración de Player y Stage, de forma que para ejecutar la simulación en Stage, bastará con ejecutar Player junto con el archivo 'prueba.cfg'. Los demás archivos de esta subcarpeta corresponden a las definiciones necesarias para que la simulación sea correcta.

Respecto a la subcarpeta pruebas, es la carpeta en la que se guardan los ficheros de prueba, los cuales contienen los datos leídos a partir de los sensores del robot Guardián. Cuando se termine la ejecución del programa 'controlrobot', en esta carpeta se guardará un fichero de texto plano con el nombre 'datosleidos.csv' por defecto.

La subcarpeta 'src' es la carpeta en la que se almacenan todos los ficheros fuente del proyecto, es decir, los archivos que conforman el programa principal, implementado en el lenguaje C++.

Fig. 64. Organización de los archivos.

Para utilizar el middleware desarrollado, basta con disponer de la carpeta 'middleware' que contiene todos los archivos necesarios. Una vez que se dispone de la carpeta 'middleware' se puede probar a ejecutar el sencillo programa de control o se puede iniciar un nuevo proyecto, partiendo del fichero 'controlrobot.cpp' que es el que contiene la lógica del programa de control. A continuación se describen las características de este fichero.

PROGRAMA DE CONTROL DEL ROBOT

El sistema de control del robot es bastante sencillo, ya que se ha incido en el desarrollo de un soporte para la lectura de los datos, más que en la implementación de sistema de control más complejo y elaborado. Por lo tanto, el control del robot actual permite mover el robot a través del teclado o de un joystick, mientras que se van leyendo los datos de los sensores y se almacenan en una lista.

Al inicio del programa de control, se realiza la conexión con el servidor Player, el cual puede estar ejecutándose en el robot Guardián o en el entorno de simulación Stage. Para realizar la conexión y activar los dispositivos sensores es necesario indicar la dirección IP del servidor Player y algunos aspectos más.

```
/**
 * Se conecta con el servidor Player y se accede a los dispositivos.
 */
void conectarServidor()
{
    // Se crea la conexión con el servidor de Player
    robot = new PlayerClient(robotIp, robotPort);

    // Se conectan los sensores
    dispMotor = new Position2dProxy(robot, 0);
    dispLaser = new LaserProxy(robot, 0);

    // Si la conexión es real, se conectan los demás sensores
    if(tipoConexion == 'r')
    {
        dispBateria = new PowerProxy(robot, 0);
        dispImu = new ImuProxy(robot, 0);
        dispGps = new GpsProxy(robot, 0);
    }

    // Se activa el motor (Odómetro)
    dispMotor->SetMotorEnable(true);

    // Se resetea el odómetro
    dispMotor->ResetOdometry();
}
```

Fig. 65. Conexión con el servidor Player.

Las variables 'robotIp' y robotPort' se definen en el fichero 'args.h', el cual se encarga de tratar los argumentos de entrada al programa. Por defecto la dirección IP de conexión es 192.2168.2.11 para la conexión con el robot Guardián, o 127.0.0.1 para la conexión con Stage. En el caso de que las direcciones IP hayan cambiado o simplemente se quiera utilizar otras, basta con indicar la dirección de conexión a utilizar mediante el argumento opcional '-h <hostname>'.

El programa de control, implementado en el archivo 'controlrobot.cpp', posee un bucle principal, en el que en cada iteración se realiza un procesamiento. Dentro de este bucle es donde debe implementarse el comportamiento del robot.

```
// En cada iteracion se ejecuta: Lectura - Logica - Accion
while(!fin)
{
    // Se leen los nuevos datos se actualizan los sensores.
    robot->Read();
    leerSensores();

    // La logica del programa de control
    actualizar(&cont);

    // Se envia el comando de accion, en este caso es mover el robot
    dispMotor->SetSpeed(cont.velocidad, -cont.giro);

    // Ya se ha realizado la accion
    cont.accion = false;
}
```

Fig. 66. Bucle principal del programa de control.

En cada iteración del bucle, se realizan tres pasos: lectura de sensores, actualización o lógica del programa de control y por último se ejecutan las acciones, que como se puede apreciar en la figura de arriba (fig. 66), en este caso se indica al robot que modifique su velocidad y su giro.

A continuación se explica con más detalle cada uno de los tres pasos que se realizan en cada iteración del bucle principal.

Lectura de sensores

La lectura de sensores es un paso importante si se desea construir un sistema autónomo o un sistema de medición de algún tipo. De lo contrario, si sólo se quiere manejar el robot, este paso puede obviarse, aunque es recomendable hacerlo.

```

/*
 * Esta funcion accede a cada uno de los sensores y lee los
 * datos que estos hayan recogido. Tambien almacena los datos
 * leido en la estructura de tipo SensorDataList para su posterior uso.
 */
void leerSensores()
{
    // Position2d (motor)
    leerSensorOdometro();

    // Laser
    leerSensorLaser();

    // Si la conexion es real y no simulada se leen los demas sensores
    // de lo contrario no, porque el simulador Stage no los soporta.
    if (tipoConexion == 'r')
    {
        // Bateria
        leerSensorBateria();

        // IMU
        leerSensorImu();

        // GPS
        leerSensorGps();
    }
}

```

Fig. 67. Primer paso, lectura de sensores.

Tal y como puede apreciarse en la figura de arriba (fig. 67), la lectura de los sensores consta de la lectura de los sensores odómetro y láser, en el caso de que se esté utilizando el programa de control en una simulación con Stage. Si por el contrario se trata de controlar al robot Guardián, además de leer los sensores odómetro y láser, se leen los sensores de la batería, la IMU y el GPS.

La lectura de cada uno de los sensores es similar, con la excepción de que los datos leídos en cada caso, corresponden a las especificaciones de cada sensor. Por este motivo se describe únicamente el proceso de lectura y almacenamiento de los datos de los sensores en la estructura de tipo lista, para uno de los sensores. Los demás siguen el mismo procedimiento.

```

/*
 * Lee los datos del sensor GPS.
 */
void leerSensorGps()
{
    float altitude, longitude, latitude;
    double tiempo;
    struct timeval tActual;
    GpsSensorData datoSensorGPS;

    try
    {
        altitude = dispGps->GetAltitude();
        longitude = dispGps->GetLongitude();
        latitude = dispGps->GetLatitude();

        // Se obtiene el tiempo actual
        gettimeofday(&tActual, NULL);
        tiempo = ((double)tActual.tv_sec + ((double)tActual.tv_usec .....

        // Se rellenan los atributos del nuevo dato leído antes de
        // introducirlo en la lista.

        datoSensorGPS.setSensorType(GPS_SENSOR);
        datoSensorGPS.setTimeStamp(tiempo);

        datoSensorGPS.setAltitude(altitude);
        datoSensorGPS.setLongitude(longitude);
        datoSensorGPS.setLatitude(latitude);

        lista->addSensorData(datoSensorGPS);
    }
    catch(PlayerCc::PlayerError e)
    {
        printf("GPS: Error: %s\n", (char *) e.GetErrorStr().c_str());
    }
}

```

Fig. 68. Función de lectura de los datos del sensor GPS.

La lectura de sensores consiste en obtener los datos que el sensor ha recogido y almacenarlos en una estructura de tipo lista para su posterior uso. Para obtener los datos, se accede a al sensor a través del dispositivo asociado a Player, que en el ejemplo de la figura (fig. 68) es 'dispGps'. A través de los métodos proporcionados por las librerías de Player, se obtiene cada uno de los valores, en el caso del GPS corresponden a: altitud, latitud y longitud.

Una vez se han obtenido los datos del sensor, se crea un objeto de la clase 'datoSensorGPS' que almacenará dichos datos de forma encapsulada y mucho más manejable. Finalmente se introducen los datos en la lista, no sin antes comprobar que los datos leídos son nuevos y no repetidos (esta comprobación se hace al insertar).

Lógica del programa

El segundo paso consiste en tratar la lógica del programa de control, es decir, la actualización del estado del programa. En cada actualización, el programa ha de realizar los cálculos necesarios para simular el comportamiento deseado. En este caso se realiza una sencilla detección y evasión de obstáculos [18]. Es en este punto donde futuros proyectos han de continuar para implementar nuevas formas de control y comportamiento.

```

/*
 * Representa la logica del programa de control. Se actualizan los
 * aspectos oportunos como en este sencillo caso, que se realiza
 * una simple deteccion de obstaculos.
 */
void actualizar(struct controlador *cont)
{
    struct timeval tActual, tUltima;

    // Deteccion de obstaculos
    detectarObstaculos(cont);

    // TODO: Hacer mas cosas, simular comportamientos, etc.
    // .....

    // Se obtiene el tiempo actual
    gettimeofday(&tActual, NULL);

    // Si no hay una accion y ha transcurrido cierto tiempo desde la
    // ultima accion, se detiene el robot. Es decir, que si por ejemplo
    // se ha pulsado la tecla W para mover el robot hacia delante, para
    // que no se quede moviendose todo el rato, transcurridos unos
    // instantes (MAX_TIEMPO_SIN_ACCION) se detiene el robot.
    if (cont->accion)
    {
        tUltima = tActual;
    }
    else if (((tActual.tv_sec + (tActual.tv_usec / 1e6)) - (tUltima.tv_sec .....
    {
        cont->velocidad = 0;
        cont->giro = 0;
    }
}

```

Fig. 69. Fase de actualización del programa de control.

Además de la detección de obstáculos, se implementa en este caso un mecanismo para detener el robot si ha transcurrido un cierto tiempo desde que se ejecutó la última acción. De esta forma, el robot sólo se mueve si se mantiene pulsada una de las teclas de dirección (W, S, A, D) o si el mueve el joystick, con lo que se asegura que el robot sólo se mueve ante las acciones del usuario.

Acciones a realizar

El último paso que se realiza en cada iteración del bucle principal son las acciones que el robot ha de llevar a cabo, como por ejemplo moverse. En este caso, dada la sencillez del programa de control, la única acción que se realiza es la de modificar la velocidad y dirección del robot, según los resultados obtenidos en la lógica del programa del paso anterior.

```
// Se envia el comando de accion, en este caso es mover el robot
dispMotor->SetSpeed(cont.velocidad, -cont.giro);

// Ya se ha realizado la accion
cont.accion = false;
```

Fig. 70. Acciones a realizar en cada iteración del bucle principal.

Las funciones utilizadas en este programa de control, como por ejemplo 'SetSpeed()' (fig. 70), se apoyan en las librerías de Player, por lo que para conocer todas las posibilidades que esta plataforma brinda al desarrollador, sólo hay que dirigirse a la página Web del Player/Stage [3].

LISTA DE DATOS DE SENSORES

El otro apartado involucrado en el control del robot, es la forma en la que se almacenan y tratan los datos leídos desde los sensores. Los datos son leídos por los sensores y posteriormente se encapsulan en objetos derivados de la clase 'SensorData' y se almacenan en una estructura de tipo lista, representada por la clase 'SensorDataList', la cual implementa diversos métodos de acceso a los datos. Estos métodos de acceso a los datos pueden modificarse para adaptarlos a las necesidades o incluso se pueden añadir nuevas formas de acceder a dichos datos, con sólo modificar el código del archivo "sensordatalist.cpp".

A continuación se describe con más detalle las principales características que poseen los archivos que representan a los datos y a la estructura de tipo lista que los almacena para su posterior uso. De esta forma, cualquiera podrá utilizarlos en un nuevo proyecto o para continuar con este mismo.

Encapsulación de los datos de los sensores

Los valores leídos por los sensores no se guardan directamente en la lista, ya que cada lectura se compone de varios valores, para el GPS por ejemplo, se necesitan los valores de la altitud, de la latitud y de la longitud. Por este motivo, se ha decidido encapsular los valores referentes a una misma lectura, en un objeto de la clase 'SensorData'. Dado que todos los datos poseen las mismas características y sólo se diferencian en el significado de los mismos, todos los datos se representan como objetos derivados de la clase 'SensorData'. Sin embargo es necesario poder identificar cada tipo de dato, por lo que los datos de cada tipo de sensor, se encapsulan en objetos de clases derivadas de la clase 'SensorData', en concreto, dependiendo del tipo de sensor del que se leen los datos, se utiliza una clase u otra, pudiendo ser:

- ⌘ OdometerSensorData
- ⌘ LaserSensorData
- ⌘ ImusensorData
- ⌘ GpsSensorData

Todas estas clases se definen en archivos separados, identificados por el nombre de la clase (odometersensordata.h, odometersensordata.cpp, etc.). Como ya se ha dicho, todas estas clases derivan de la superclase 'SensorData', la cual define los atributos comunes para todos los datos, que son:

- ⌘ Información del sensor
- ⌘ Marca de tiempo
- ⌘ Valores en bruto
- ⌘ Valores procesados (si procede)

Cada una de las clases específicas se encarga de almacenar los valores en los vectores destinados a ello, dependiendo de si se trata de valores en bruto o de valores procesados. Así pues, el desarrollador únicamente ha de preocuparse de realizar las llamadas correspondientes proporcionadas por cada una de estas clases y olvidarse de cómo se almacenan realmente los valores. La gran ventaja de utilizar esta forma de encapsulamiento, es la facilidad que se tiene para manejar los datos.

Si en un futuro se instalan nuevos sensores en el robot, se puede añadir una nueva clase que represente al nuevo tipo de sensor. Para hacerlo, no hay más que fijarse en cómo se definen el resto de clases ya implementadas y seguir las mismas directrices.

La estructura en forma de lista

La estructura que se utiliza para almacenar los datos leídos desde los sensores, es básicamente una lista. En esta estructura se guardan objetos de tipo 'SensorData', tal y como se ha explicado con anterioridad. La clase 'SensorDataList' es la encargada de representar esta estructura con forma de lista, además de definir una serie de atributos y funciones que permitan gestionar los datos de manera más eficiente.

Las funciones definidas en la clase 'SensorDataList' permiten obtener los datos de diferentes formas, por ejemplo se pueden obtener los datos de un tipo de sensor exclusivamente.

Cabe destacar que al introducir nuevos datos en la lista, se realiza antes una sencilla comparación, la cual ha de mejorarse en un futuro. Esta comparación comprueba si los datos a insertar son datos nuevos, o si por el contrario se trata de datos repetidos. Este es un problema derivado de la asincronía entre los sensores, explicado a lo largo de todo el documento del proyecto.

OTROS ASPECTOS DEL CONTROL DEL ROBOT

Para clarificar aún más los conceptos relacionados con el middleware desarrollado en este proyecto, se presenta a continuación el programa de control utilizado a modo de ejemplo. Este programa permite mover el robot de forma manual, a través del teclado o de un joystick estándar, a la vez que se van recogiendo los datos medidos por los sensores del robot y se van almacenando en la estructura de tipo lista que implementa el middleware. No se muestra todo el código del ejemplo (se incluye en la carpeta de archivos adjuntos a este proyecto), ya que la mayor parte ya ha sido comentada, por lo que en este punto se hace especial hincapié en el control del robot y se deja de lado la gestión de los datos de los sensores y la organización del programa.

Definición de macros variables

El primer paso en el programa de control consiste en definir las variables y macros de las que se va a hacer uso. En este ejemplo se definen entre otras las variables relacionadas con la configuración del robot y de su funcionamiento. Concretamente se definen las teclas del teclado que van a utilizarse, la velocidad y giro máximos relacionados con el movimiento del robot y la escala del joystick, la cual permite realizar una normalización sobre los valores obtenidos cuando se utiliza el joystick.

```

#define KEYCODE_W 0x77
#define KEYCODE_S 0x73
#define KEYCODE_A 0x61
#define KEYCODE_D 0x64

#define MAX_VELOCIDAD 1
#define MAX_GIRO DTOR(100)

#define MAX_TIEMPO_SIN_ACCION 0.2
#define ESCALA_JOYSTICK 32767.0

```

Fig. 71. Definición de las macros globales.

Una vez definidas las variables, o macros en este caso, que van a utilizarse a lo largo del programa de control, se definen otras variables y estructuras globales como las que se muestran en la siguiente figura (fig. 72):

```

// Esta estructura 'controlador' se utiliza como ayuda
// para actualizar la velocidad y el giro del robot.
struct controlador
{
    double velocidad;
    double giro;

    //se utiliza para saber cuando se pulsado una tecla
    // y por tanto hay que enviar un comando.
    bool accion;
};

```

Fig. 72. Variable controlador.

La estructura “controlador” permite leer y modificar los valores de la velocidad y giro de una forma sencilla. Esta estructura contiene además contiene un campo llamado “acción”, el cual se utiliza para conocer el estado actual del control del robot, lo que significa que cuando “acción” es verdadero, es que el usuario ha pulsado una tecla o ha movido el joystick, por lo que es necesario procesar los nuevos valores de velocidad y giro y enviárselos al robot. Si “acción” es falso y ha pasado un cierto tiempo (MAX_TIEMPO_SIN_ACCION) entonces sea cual sea el estado del control del robot, se restablecen los valores de velocidad y giro a cero y se detiene el robot. Esto es una medida de seguridad.

El siguiente paso es la definición del método “main” del programa, que es el punto de entrada de la aplicación.

Gestión del control o movimiento del robot

Para mover el robot el usuario puede utilizar el teclado o el joystick. Por defecto se utiliza el joystick, pero si no se dispone de él, se utiliza el teclado.

El uso del joystick es muy sencillo, basta con moverlo en la dirección en la que se quiere mover el robot y dependiendo de la amplitud del movimiento, la velocidad con la que el robot se mueva será mayor o menor. La implementación del joystick se encuentra en los archivos “joystick.h” y “joystick.cpp”. El joystick se trata como un descriptor de fichero del cual se pueden leer datos, por lo que su codificación no es muy compleja. Un aspecto a tener en cuenta es la definición de la macro que especifica la ubicación del dispositivo joystick, dentro del archivo “joystick.h”.

```
// Por defecto el joystick se encuentra en
// esta ruta, comprobar si da error.
#define JOY_DEV "/dev/input/js0"
```

Fig. 73. Ubicación del joystick en Ubuntu 9.10.

Para hacer uso del joystick o del teclado, es necesario comprobar cada cierto tiempo si el usuario ha realizado alguna acción. Por este motivo, la gestión del joystick y del teclado se realiza en hilos de ejecución separados, de forma que no interfieran en la lógica de control del robot, además de que la comprobación de los cambios en el teclado y en el joystick ha de realizarse con mayor rapidez. Así pues, la gestión del teclado y del joystick se realiza de manera independiente al bucle principal de control del programa.

Antes de iniciar el bucle principal de control, se inicializa la gestión del joystick o del teclado (encaso de que el joystick no esté disponible). Cabe destacar, que el joystick se encapsula en la clase Joystick, definida en los archivos indicados anteriormente y la cual proporciona unos sencillos métodos de acceso al dispositivo joystick, entre los que destacan:

- ⌘ Abrir el joystick.
- ⌘ Cerrar el joystick.
- ⌘ Leer del joystick (proporciona los valores de los ejes X e Y).

```

joy = new Joystick();

// Si el joystick esta conectado, se utiliza,
// si no se usa el teclado.
if (joy->abrirJoystick() == 0)
{
    // Si todo va bien, se inicia el control del joystick,
    // al que se le pasa una variable de tipo controlador,
    // para que sean visibles las modificaciones de la velocidad y giro.
    printf("Abierto el joystick...\n");
    pthread_create(&hilo, NULL, &procesarJoystick, (void*)&cont);
}
else
{
    // Si todo va bien, se inicia el control del teclado,
    // al que se le pasa una variable de tipo controlador,
    // para que sean visibles las modificaciones de la velocidad y giro.
    printf("Abierto el teclado... \n");
    pthread_create(&hilo, NULL, &procesarTeclado, (void*)&cont);
}

```

Fig. 74. Creación de hilos separados para la gestión del input.

Como puede apreciarse en la imagen superior (fig. 74) se crea un hilo de ejecución separado del principal, que se encarga de gestionar el joystick o el teclado en el caso de que el joystick no se pueda abrir. Este trozo de código se encuentra justo antes del bucle principal de control.

La función “procesarJoystick” (fig. 75) que recibe como parámetro un puntero a la estructura de tipo “controlador”, consta de un bucle infinito en el cual en cada una de las iteraciones se lee el buffer del dispositivo joystick, y se actualiza la información del controlador, velocidad y giro, en consecuencia. No hay que olvidarse de que los datos proporcionados por el joystick han de ser normalizados utilizando la escala definida al principio del programa, ya que de otro modo, los valores de velocidad y giro no se corresponderían con lo que el usuario realmente quiere hacer.

La función que controla el teclado sigue exactamente la misma estructura, salvando las diferencias.

```

/*
 * Funcion que se ejecuta en un hilo y que
 * procesa las acciones del joystick.
 */
void* procesarJoystick(void *arg)
{
    double velocidad, giro;
    int ejeX, ejeY;
    struct controlador *cont;

    // Se apunta al controlador principal para cambiarle
    // los parametros de velocidad y giro.
    cont = (struct controlador*)arg;

    // Bucle infinito para capturar en cada iteracion
    // la accion del joystick.
    for(;;)
    {
        // Se lee info del joystick
        joy->leer(&ejeX, &ejeY);

        // Se normaliza con la escala del joystick para
        // obtener valores correctos.
        velocidad = -ejeY / ESCALA_JOYSTICK;
        giro = -ejeX / ESCALA_JOYSTICK;
        cont->accion = true;

        // Si se ha realizado una accion,
        // se normalizan los valores.
        if (cont->accion == true)
        {
            cont->velocidad = velocidad * MAX_VELOCIDAD;
            cont->giro = giro * MAX_GIRO;
        }
    }
}

```

Fig. 75. Función que gestiona el joystick.

Después de crear los hilos de ejecución paralelos al programa principal, se inicia el bucle de control del programa, en el cual se leen los datos de los sensores, se aplica la lógica y el comportamiento y se ejecutan las acciones pertinentes, tal y como se ha explicado anteriormente.

Ejecución de los ejemplos

Los programas de control desarrollados para ejecutarse con el robot Guardián o en el simulador proporcionado por Stage, han de ser compilados previamente, como cualquier otro programa implementado en C/C++. En el caso del ejemplo del middleware de este proyecto, basta con acceder a la carpeta “middleware” proporcionada junto con este documento y ejecutar el comando “make” desde un terminal. De este modo se creará el ejecutable “controlrobot” en la carpeta “middleware/bin”.

La ejecución de cualquier programa de control requiere que exista un servidor Player corriendo en el robot Guardián, o en el mismo PC que el programa de control si se trata de una simulación con Stage. En el caso del programa de este ejemplo, suponiendo que existe un servidor Player corriendo en el robot Guardián en la dirección 192.168.2.11 (escuchando por el puerto 6665), la ejecución del programa de control, que se encuentra en la ruta “middleware/bin” sería de la siguiente forma:

⌘ `./controlrobot -m r`

Con este comando se ejecuta el programa de control del robot en modo real, nótese la opción “-m r” que indica que el entorno es el real. Si por el contrario se desea utilizar el programa dentro de una simulación, primero hay que arrancar el servidor Player dentro del mismo PC y luego ejecutar el programa de control en modo simulación (-m s). Partiendo desde la carpeta “middleware” se haría de la siguiente forma:

⌘ `player config/prueba.cfg` (en un terminal).

⌘ `./controlrobot -m s` (en otro terminal y desde la ruta “middleware/bin”).

